Java Version

VI SDK Programming:

Automating, Integrating, and Extending the VMware Cloud OS

APRIL 14TH – 16TH, 2009 WALT DISNEY WORLD DOLPHIN ORLANDO, FLORIDA

09 vmware partnerexchange

Contents

Instructors

David Deeths- Senior Technical Alliance Manager, VMware (Team Captain)

Steve Jin - Sr. Member of Ecosystem Engineering Technical Staff, VMware

Rajesh Kamal - Member of Ecosystem Engineering Technical Staff, VMware

John Kennedy - Senior Technical Alliance Manager, VMware

Balaji Parimi - Sr. Member of Ecosystem Engineering Technical Staff, VMware

Paul Vasquez - Staff Technical Alliance Manager, VMware

Alton Yu - Technical Alliance Manager, VMware

About This Lab

This lab will provide hands-on training for integrating programs with a VMware® environment. The goal of this lab is to provide all the tools that you need to extend your programs to automate, integrate, and extend VC. For *VI SDK Programming: Automating, Integrating and Extending the Cloud OS*, participants can choose to use the C# or Java versions of the VI SDK to monitor and manage the VMware Infrastructure. You will walk away from the lab with a better understanding of VMware Infrastructure, resources for using the VI SDKs, and a variety of simple code examples for performing common administrative tasks. Exercises will cover performing tasks, examining virtual machine attributes, performing actions on many virtual machines at a time, and exporting performance data. This lab does not require programming or scripting experience.

Introduction

Welcome to the ***VI SDK Programming: Automating, Integrating and Extending the VMware Cloud OS*** lab at Partner Exchange Spring 2009. This lab will demonstrate some of the capabilities of the VI SDK. The SDK consists of programming interfaces that allow anyone to write programs that can monitor and manage any aspect of VMware Infrastructure. The SDKs allow programs to connect to a Web service interface from either VirtualCenter or a VMware ESX server.

This hands-on lab consists of two manuals, one for participants who are programming with the C# SDK and another for participants who are programming with the Java SDK. *Please take a moment to verify that you are using the appropriate manual*. The lecture portion of the hands-on lab will be the same for both toolkits.

This lab will include a short presentation as well as hands-on experience writing programs for administration of VMware Infrastructure. A programming background is required for this lab, but participants do not necessarily need experience with Java, C#, or web services if they are comfortable with programming concepts; the lab will stress VMware Infrastructure programming concepts and architecture more than programming techniques. The lab will attempt to minimize the need to understand language-specific issues. The background sections of this handbook will concentrate on the theory and data structures involved in SDK programming, and the hands-on programming exercises will focus on the specific implementation.

As part of this lab, you will write or modify C# or Java methods to demonstrate key SDK functionality. These hands-on exercises will include an examination of methods to gather virtual machine configuration information and performance history and to reconfigure a virtual machine. Experience with a programming language and familiarity with virtual machines, VMware ESX server, and VirtualCenter is required.

Because of the lab's security configuration, you cannot create virtual machines in the root folder. In this lab, you must create virtual machines in folders and resource pools that have been configured for you. Your folder name is `studentN` where `N` is your student ID.

Key Concepts

There are a number of important key concepts for using the VI SDK. The sections below outline some of the key areas to understand.

Interfacing with VMware Infrastructure

The VMware Infrastructure API (VI API) is the most commonly used API and is for developers who want to create client applications that can manage, monitor, and maintain VMware Infrastructure (made up of VMware ESX servers, VirtualCenter instances, and VMware Server systems that are managed by VirtualCenter). The VMware Infrastructure SDK (VI SDK) is based on the VI API, as are the VI Toolkits. The SDK and toolkits simplify access to the VI API.

The VI API works with any language that supports Web services. The API includes pre-built Web Services Description Language (WSDL) stubs that simplify the process of using the API with Java (using Apache Axis) and C# (using Microsoft .NET). The pre-built stubs, libraries, and development tools for Java and C# form the VI SDK. The VI SDK essentially exposes the VI API Web service directly to the appropriate language; all the objects and the API and SDK methods are the same and little abstraction exists.

Developers can create their own stubs using the webservices framework of their choice, or they can access the web service directly by sending SOAP requests.

In addition to the VI SDK, Perl and Windows PowerShell VI Toolkits allow interfacing to the VI API. The toolkits allow the same capabilities as the VI SDK but have a higher degree of abstraction. The level of abstraction in the toolkits generally makes actions on the VI API easier to perform than does the VI SDK. However, this abstraction might hide some of the fine-grained control and performance optimization that the VI SDK provides.

<table>
<tr><th>Underlying API</th><th>Specific Method</th><th>Development Language</th></tr>
<tr><td rowspan="5">VI API</td><td>VI API Web Service (direct access to XML)</td><td>Direct XML over HTTP or through a Web services development environment</td></tr>
<tr><td rowspan="2">VI SDK</td><td>Java</td></tr>
<tr><td>C#</td></tr>
<tr><td rowspan="2">VI Toolkits</td><td>VI Perl Toolkit</td></tr>
<tr><td>VI Toolkit (for Windows) in Powershell</td></tr>
</table>

In addition to the VI API, VMware provides several other interfaces for working with the VMware virtual infrastructure. These interfaces represent different APIs, each intended for different developer uses.

The following table clarifies some of the differences between the VI API (including the VI SDK and VI Toolkits) and the other VMware APIs.

API Category	API Name	Intended Use
Management	VI API (used by VI SDK and VI Toolkits)	Provides VMware Infrastructure management and monitoring of virtual machines, hosts, VMware ESX configuration, and other resources managed by VMware ESX and VirtualCenter
	CIM SMI-S	Uses the Common Information Model (CIM) standard to report the organizational structure of virtual machines and associated storage
	CIM SMASH	Uses the CIM standard to monitor and control the underlying hardware running VMware ESX
Virtual Disk Access	VDDK	Can access virtual disks remotely, perform actions on VMware Virtual Machine File System (VMFS), and move virtual disk files on and off the VMware VMFS
Virtual Machine Automation	VIX API	Automates operations inside virtual machines like starting/stopping processes and moving files in and out of a VM
Guest	Guest SDK	Enables read-only access to VMware Infrastructure 3 state and performance data from inside a guest operating system
Legacy	VmPerl	Deprecated interfaces that should not be used in new programming projects; do not confuse the old VmPerl with the newer VI Perl Toolkit
	VmCOM	

Versions of the VI API

VI API version numbers are derived from the version number for the corresponding VirtualCenter release, not the version number of the VMware ESX release. So VI API 2.0 covers all the features in VirtualCenter 2.0, regardless of VMware ESX version, and VI API 2.5 covers the features in VirtualCenter 2.5.

The version numbers of the toolkits are independent of the VI API version. The VI Perl Toolkit version 1.0 uses VI API 2.0, and the VI Perl Toolkit 1.5 and 1.6 use VI API 2.5. The VI Toolkit (for Windows) 1.0 uses VI API 2.5.

Connecting to VirtualCenter and VMware ESX

The VI API operates as a Web service. The toolkit clients connect over HTTP or HTTPS and send requests, in the form of XML documents, to the server. The server responds with the results, also in the form of XML documents. The toolkits hide this data transfer from the programmer and represent the objects and associated XML as local Perl or Windows PowerShell objects.

Connecting to the Web service requires a login and authorization. You can use any user accounts with the VI SDK to access VMware Infrastructure. When connected through the toolkits or VI API, users have the same permissions that they would have when using the VMware Infrastructure Client.

Users can connect to either VirtualCenter Server or a VMware ESX server. The structure of the scripts and commands is the same with either server. However, VirtualCenter provides several functionalities that are not available in VMware ESX, such as VMware VMotion and VMware Distributed Resource Scheduler (VMware DRS). In addition, the inventory structure is a bit different when connecting through VirtualCenter because VirtualCenter has more layers of abstraction (data centers, clusters, and folders). Because of this, it is best to always operate through Virtual Center when it is available in an environment.

For a complete list of the API operations supported on ESX server and VirtualCenter Server, refer to the SDK v2.5 Programming Guide, Appendix B (http://www.vmware.com/support/developer/vc-sdk/visdk25pubs/visdk25programmingguide.pdf).

Objects in the VMware Infrastructure versus objects in a script

Two types of objects exist in the VI SDK: *managed objects* and *data objects*. A managed object represents either a physical item in inventory (such as a VirtualMachine or a HostSystem), an inventory grouping (such as a Folder or Datacenter), or a service involving inventory items (such as the performance management service or the login session management service). In the scripting toolkits, managed objects are sometimes referred to as server-side objects.

Because managed objects represent objects that make up the structure of VMware Infrastructure, they exist only on the server, not on the client machine that is running a VMware Infrastructure script. VMware Infrastructure scripts can interact only indirectly with managed objects, by using representations of those objects (called *managed object references*) and objects in the script (called *data objects*). Managed object references essentially specify the ID of the managed object.

Data objects are used by VI scripts and represent parts of a managed object and appropriate methods to interact with managed objects on the server. A data object represents information about a managed object. Any object in the VI API that does not represent an object that exists on the server is a data object. The data about managed objects that has been transferred to the client is also represented by data objects. Data objects are typically composed of other data objects, references to managed objects, faults, or other basic data types. Information about the proper variable names, properties, and methods needed to extract the data from VMware Infrastructure are described in detail in the VI SDK documentation

and can also be found in the Managed Object Browser interface (described below). In many cases, creating or modifying a managed object requires the use of a data object storing the appropriate specifications. For example, a **`PropertyFilterSpec`** represents a specification that defines a **`PropertyFilter`** managed object.

In contrast to managed objects, Data objects can be passed by value between the client and the web service.

When using a VI Toolkit, the distinction between managed objects and data objects is informal. In contrast, this distinction is very important when using the VI SDK.

Managed Object References

Managed Object reference represent the a unique ID for a remote object. While it is sometimes convinient to think of managed object references as pointers or java/C# object references, the actual implementation is that these are simply ID numbers used by VC and ESX to uniquely identify particular objects on the server. The managed object references in no way refer to memory locations on the VC or ESX server.

Because you cannot directly manipulate the managed objects (which represent actual items in the VMware Infrastructure) in a script, any time you want to affect one of those objects, you must get the object's reference ID and use that ID as input to the function that you are using to manipulate the object.

Managed object references can be obtained in several ways:

• It can be constructed (this only works for the **ServiceInstance** object, which is the root of the object hierarchy and therefore needs to be obtained directly)

• It can be the result of an invoked operation that returns a Managed Object Reference

• It can be obtained via an accessor method (if a data object type contains a managed object reference as one of its properties)

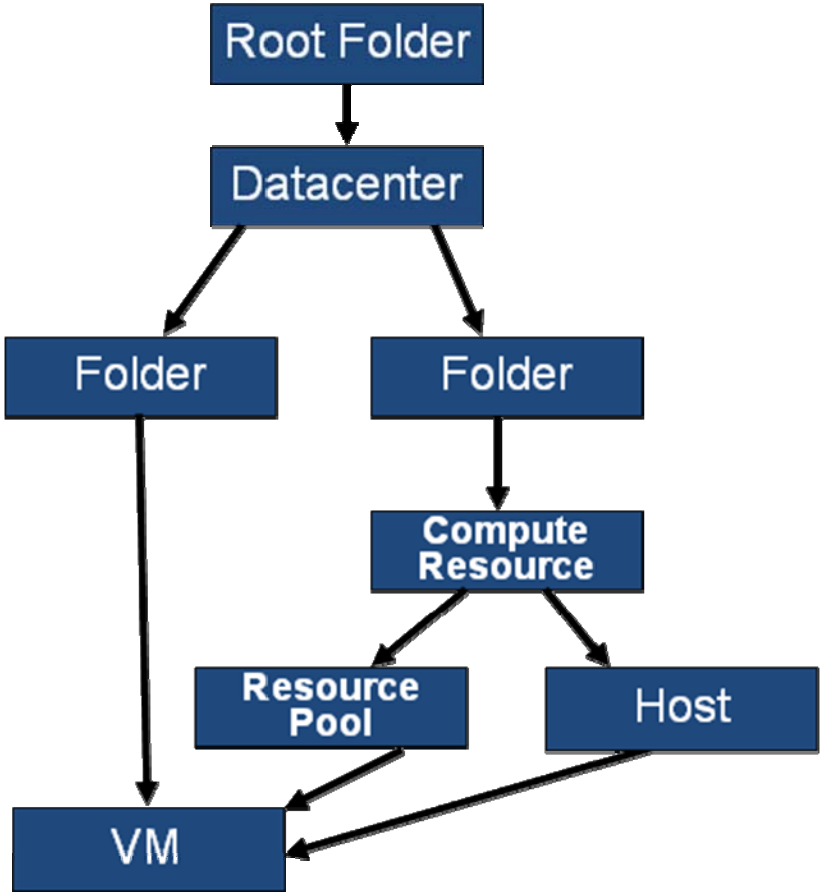
• It can be found using the property collector (this is typical for managed entities that are in the inventory tree, such as virtual machines and hosts), or for managed object references that are properties of another object

The object hierarchy

Objects in VC are organized as a hierarchy. The illustration below shows a small example of part of the VC hierarchy. The two forks correspond to the two views of VMs in the VI client ("Hosts and Clusters" and "Virtual Machines and Templates").

The diagram shows how objects, such as Datacenters, Hosts, Resource Pools and VMs are categorized.

Some objects, like folders and resource pools, can also be nested. For some of the more advanced exercises, you will need to understand the VI Inventory structure, but there are many sets of tools available for searching.



Using the documentation

When using the VI SDK, you should be aware of a few important sources of information. The following list describes the most useful toolkit documentation, as well as general SDK documentation that might be helpful in certain exercises. All the referenced documents are in the `Documentation` directory of the lab clients.

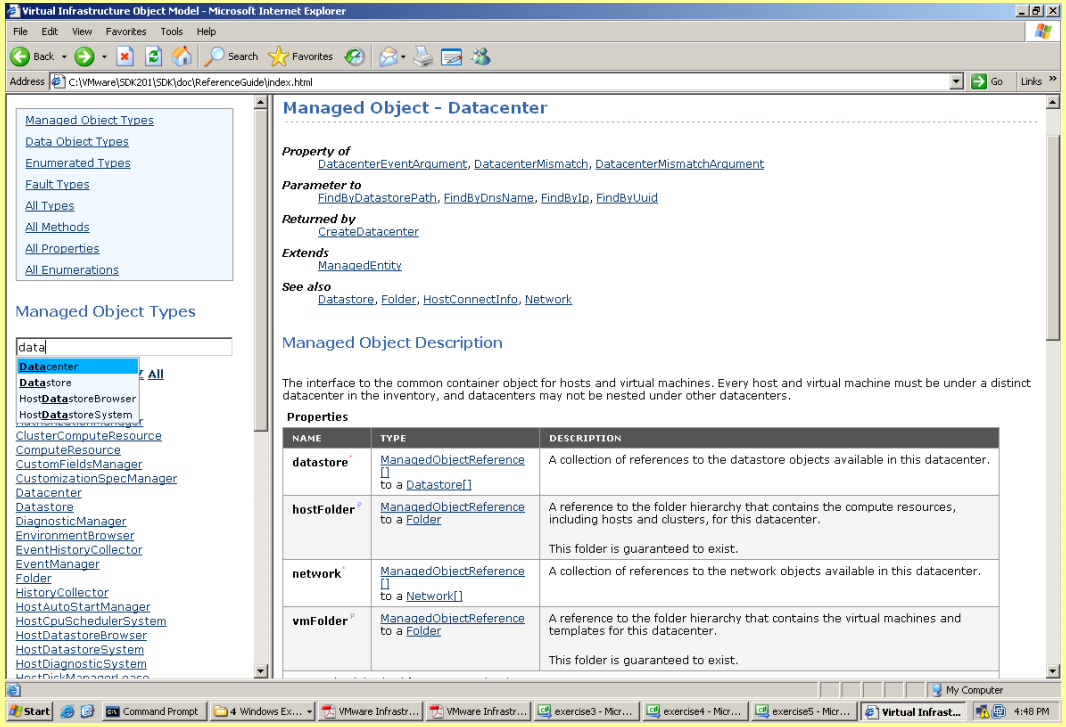
Regardless which toolkit you are using, the following references are important:

- The *Developer Setup Guide* covers how to get started with the VI SDK.
- The *VI SDK Programming Guide* covers the details of programming with the VI SDK. Although you do not need to understand the details of SDK programming to use the VI Toolkits, the appendices of this guide contain important information not found elsewhere.
    - In VI SDK 2.0 and 2.5, Appendix A of the *VI SDK Programming Guide* describes the performance counters that are available for each object in VMware Infrastructure, as well as which sorts of statistics and rollups are provided for each counter. If you plan to gather performance information from VMware Infrastructure, you will need the information in this appendix.
      In VI SDK 4.0, this is instead covered under the "Performance Manager" Managed Object section of the VI SDK Reference Guide.
    - Appendix B of the *VI SDK Programming Guide* describes the privileges required to perform any type of action on a particular VMware Infrastructure object. This appendix also describes which actions require VirtualCenter and which will work on VMware ESX. Although the methods used are specific to the VI SDK and do not always map easily to the VI Toolkits, this is the best reference for trying to understand which privilege is needed for different sorts of tasks.
- The *VI SDK Reference Guide* covers the detailed descriptions of every object available in VMware Infrastructure. The *VI SDK Reference Guide* is very complex (more than 1,300 HTML pages, with most pages including many methods and properties), and the amount of information available in the guide is huge. This is the essential tool for understanding the various properties of the objects in the VI SDK.

Understanding the *VI SDK Reference Guide*

The *VI SDK Reference Guide* (version 2.5) is available online at http://www.vmware.com/support/developer/vc-sdk/visdk25pubs/ReferenceGuide/index.html and in the *Documentation* directory on the client machine for this lab. The other VI SDK documentation covers using the Web services and the theory of the VI SDK. The *VI SDK Reference Guide* is a hyperlinked, highly interactive guide that covers the various managed objects, data objects, and enumerations used in the VI SDK. The easiest way to use the guide is to select which sort of VI SDK feature you are looking for (typically a managed object, data object, enumeration, or fault type) from the links at the left, and then type the object name in the search field. Because the search field is auto-completing, you can use it to find objects whose names you do not know by searching for key terms.

The following figure shows the *VI SDK Reference Guide* in action:

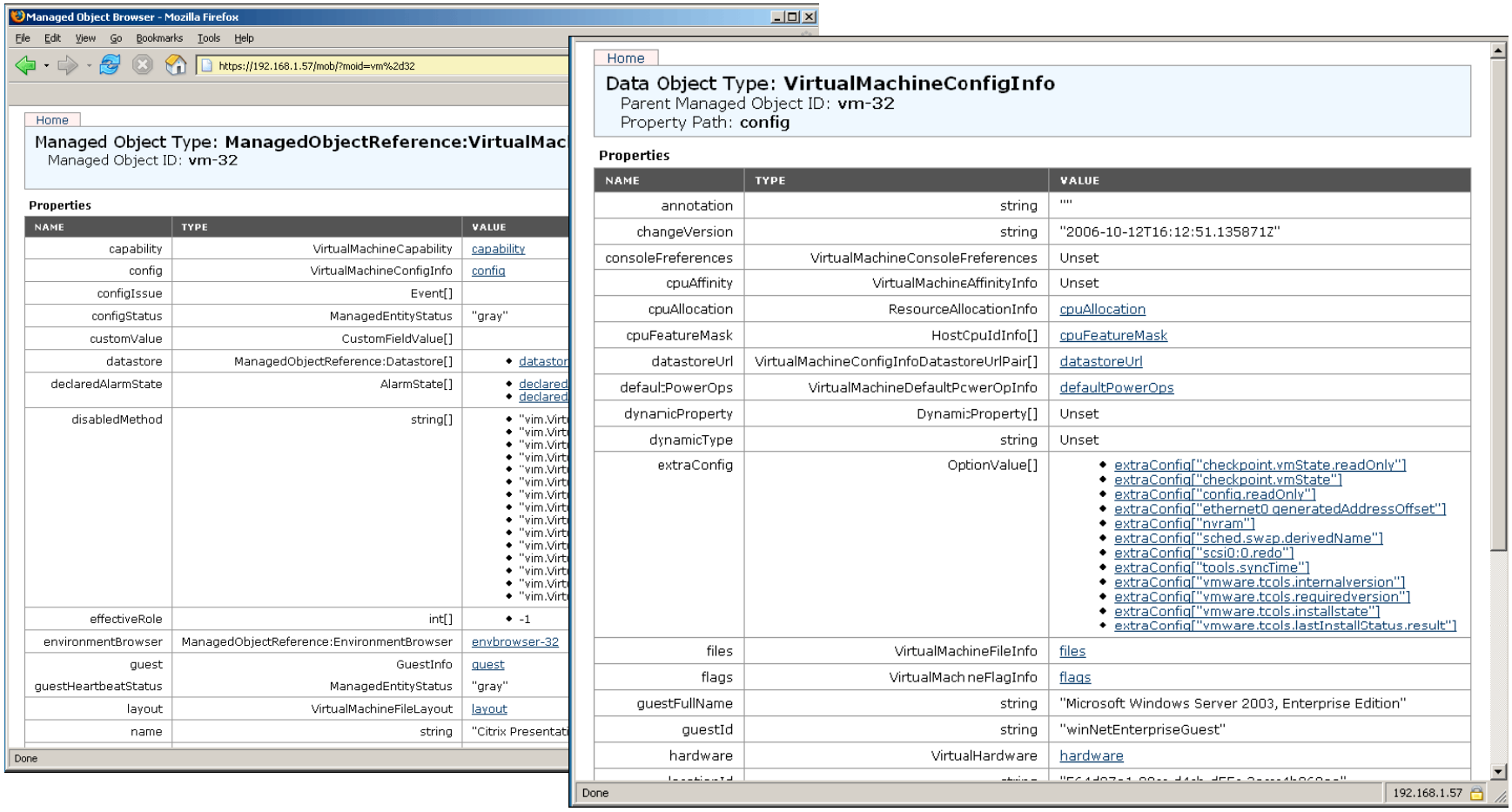


. .

Using the Managed Object Browser (MOB)

Besides all the SDK documentation, a wonderful tool for understanding all the managed objects in the VI SDK is the Managed Object Browser (MOB). This tool allows you query the target environment 'live'. You can connect to the MOB of any running ESX or VC system and interact directly with the managed objects in the SDK, providing a good way of understanding these programmatic concepts on a running server. This can be reached by pointing your browser to `https://<your_vc_server>/mob`. The ability to read, write, and manage objects uses the same roles for the MOB as for the VI SDK and for VC/ESX, so user privileges remain constant no matter what the connection method. In some user environments, the MOB may be disabled to force all administrative functions to go through other methods.

The following screenshot shows the MOB running on a live system:



The Development Environment

The VM used for this lab is setup with all the necessary components to perform SDK development in Java or C# on a Windows environment (there are other tools for developing using other languages and platforms, as described in the documentation). This includes the Microsoft .Net framework (downloaded from Microsoft), the SDK framework (downloaded from http://www.vmware.com/download/download.do?downloadGroup=VC-SDK), Visual Studio for C# development, and Eclipse for Java development. The documentation at `http://www.vmware.com/support/developer/vc-sdk/SetupGuide.pdf` explains how to setup an environment for VI SDK development. The examples used in this lab and the documentation of the specific exercises will be available from the VMworld site where you downloaded this file.

After you login to your Windows XP VM, double click the "Java Lab" or "C# lab" icon on the desktop. This will start the IDE being used for this lab.

Libraries used in this Lab

A number of libraries of common Java and C# functions are used for the lab.

Java developers will use the following libraries for this lab:

java.util.*

java.net.*

java.io.*

java.text.*

com.vmware.vim25.* (this is the library containing the VI API web service stubs)

C# developers will use the following libraries for the lab:

System

System.IO

System.Globalization

System.Web.Services.Protocols

System.Net.Security

System.Security.Cryptography.X509Certificates

Vim25Api (this is the library containing the VI API web service stubs)

Java Quick Reference

This section provides a quick reference on Java syntax and usage for folks that may be a bit rusty on their Java. Users comfortable with Java should skip to the next section.

Statements, blocks, and control loops

Every statement in perl is an expression, optionally followed by a modifier. Each statement is terminated by a semi-colon.

Execution of expressions can depend on other expressions using one of the modifiers **if**, **while**, or **try** for example:

Statements can be combined to form a BLOCK when enclosed in **{}**. Blocks may be used to control flow. There are a few basic conditionals and loops in Java, as described below.

An "if" statement represents a conditional. The else if and else parts are optional. Zero or more "else if" parts contains code to execute if particular expressions evaluate as TRUE. The single "else" part executes if no other "if" or "else if" expressions evaluate as TRUE.

```
if (EXPR)  {BLOCK}
else if (EXPR) {BLOCK}
else if (EXPR) {BLOCK}
else {BLOCK}
```

Java also includes a conditional operator that provides a way to simplify the syntax of the if/else statement. The statement:

```
value = ( UserSetValue ? usersValue : defaultValue
```

is equivalent to

```
if (UserSetValue)
   value = usersValue;
else
   value = defaultValue
```

A "while" statement represents a loop depending on the true evaluation of an expression. The continue portion is optional and will execute at the end of the last loop.

```
while (EXPR) {BLOCK}
continue {BLOCK}
```

A try statement represents an action that should be executed, but may result in an exception. An exception is a way for java methods to indicate unexpected error conditions. The exceptions details are passed as Exception objects. The catch portion of the code is executed if an exception occurs and generally cleans up after the failure and prints an error. The "finally" portion of the code represents code that should be executed after either the successful try or the unsuccessful try triggering the catch code. Typically these sections reinitialize variables or close open files.

```
try {BLOCK}
catch (Exception type identifier) {BLOCK}
catch (Exception type identifier) {BLOCK}

finally {BLOCK}
```

Functions

Functions in Java belong to classes and are usually referred to as methods. In the text, we will usually reserve the term methods for referring to the webservice methods and use the term functions to refer to the methods/functions that are defined in each program.

Operators

The following tables describe some of the operators available in Java.

Comparison & Equality Operators

Comparison	Numeric

Equal	==
Not equal	!=
Less than	<
Greater than	>
Less than or equal	<=
Greater than or equal	>=

Operators

+ -	*** / %**	Addition, subtraction, multiplication, division, remainder (modulo)
& |	**^**	Bitwise AND, bitwise OR, bitwise exclusive OR
+		Concatenation of two strings
||		Conditional OR (valid for Boolean values)
&&		Conditional AND (valid for Boolean values)
++	**--**	Auto-increment, auto-decrement

Hands-on Lab Overview

The hands-on lab portion of this class consists of several exercises. These exercises are divided into primary and optional exercises. The primary exercises teach the key concepts necessary to perform useful work by using the VI SDK. The optional exercises are more advanced and combine the concepts from the primary exercises in novel ways or apply those concepts to new VMware Infrastructure objects and features.

Because of the huge variety and scope of the VI API, this lab focuses on teaching the core concepts, while providing the opportunity to learn some variations and advanced uses of concepts.

Most of the primary exercises include some variations that are not part of the main exercise. You might have difficulty finishing the lab in the allotted time if you try all the variations; consider skipping them unless you have a strong interest in a specific topic. You will be able to try these variations later, outside the lab session.

In addition to the primary exercises and variations, some optional exercises provide an opportunity to work with different objects and features. You will have difficulty completing many of these advanced exercises in the time provided, so concentrate on the one or two that are most interesting to you.

A Security Note

In this lab, all user names and passwords will be provided on the command line. Keep in mind that storing passwords in clear-text files or calling programs with passwords as arguments represents a security risk. We do so here to make the code easier to understand and modify and to clarify the dynamics of how this information is processed.

Hands-on Exercise 1: Using the VI SDK and Connecting to the VI API

Exercise 1 teaches how to setup a simple program template for use with the VI API, how to include the VI API libraries, and how to connect with the VI API.

1a: Exploring the VI SDK

Your instructor will walk you through some exploration of the VI SDK.

1b: Creating a Simple VI SDK Program: Connect.java

The purpose of this exercise is to create a simple program template you can use to connect to the VI API Web Service. The later exercises will use the methods in this exercise to connect to the web service and make VI API calls.

Step 1: Understand Connecting to the VI API Web Service

In any VI SDK program, the following actions must be accomplished:

1. Connect and authenticate to the target Web Service
    a. Open an http/s connection to the web service port
    b. Get the service content associated with the web service over the http port
    c. login and connect to the web service
2. Retrieve information (via references to objects)
3. Process the information / watch for events / act on them
4. Disconnect from the Web Service

The **ServiceInstance** is the central point for achieving steps 1, 2, and 4. Step 3 is accomplished either by manipulating data objects in your program or by invoking some method of ServiceInstance to accomplish it on the server.

The critical thing to understand during the connection process is the difference between the http connection and the web service connection. Any client connecting to the http/s port hosting the webservice has an http connection, but it only is connected to the VI API web service after it logs in and begins sending SOAP messages.

Once connected, all communication with the VI API is accomplished by communicating with the ServiceInstance managed object. For any VI SDK application, a reference to a ServiceInstance managed object must be created during the login process. This lets the program get a reference to a SessionManager managed object, one of the parameters for the Login

operation. The ServiceInstance does not have directly accessible properties because reading the properties of a managed object requires the help of a PropertyCollector managed object reference, which itself is a property of the ServiceContent data object type. To access these properties, the ServiceInstance provides the RetrieveServiceContent operation which returns the ServiceContent data object type.

In the SDK, any methods being called on managed objects are actually called on the ServiceInstance (since it is the only managed object that exists both on the server and the client). The syntax of these calls is to call the method on the service instance with the managed object being modified as the first argument. The VI SDK Reference Guide describes methods as belonging to specific managed objects such as VMs. For instance, the VirtualMachine managed object has the method PowerOnVM_Task(). For a virtual machine with a MORef stored in the variable VM_MORef, it might seem that the following syntax is correct:

Incorrect pseudocode	VM_MORef.PowerOnVM_Task();

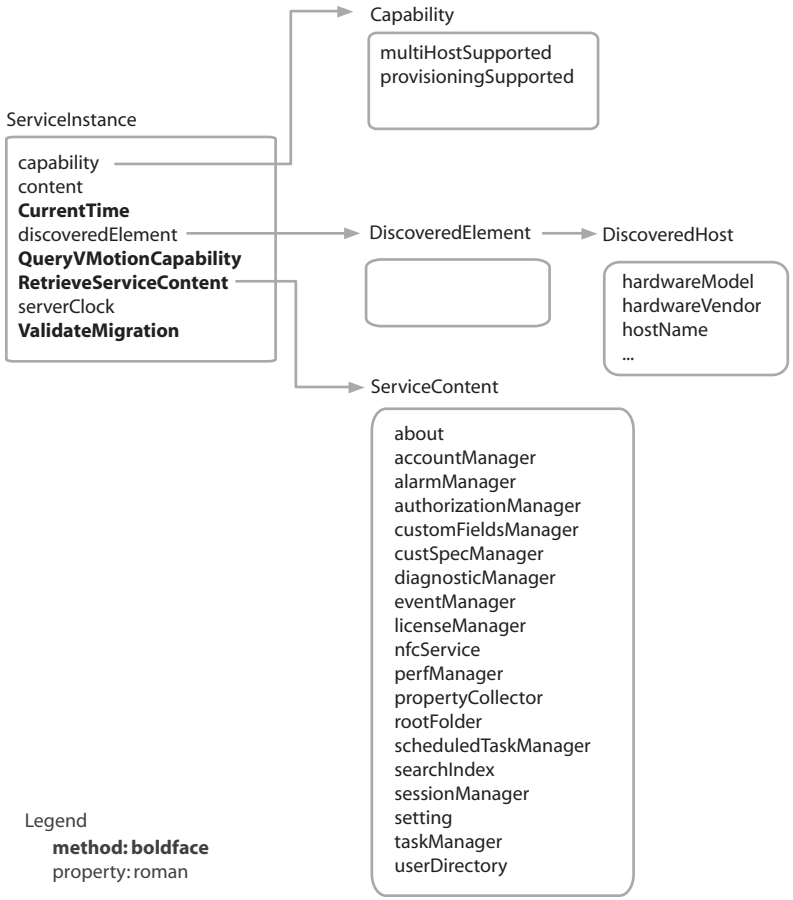
However, keep in mind that the MORef is simply an ID number for a VM. In order to actually interact with the VM itself, a connection with the service instance is required. The actual syntax to make calls on managed objects in the SDK is to make the method call on the service instance, and then to specify the MORef (ID) of the remote managed object you want to affect the method on.

Correct pseudocode	SERVICE_INSTANCE.PowerOnVM_Task(VM_MORef);

If a managed object method has arguments, the additional arguments always appear after the MORef being acted upon on the remote server.

The VI SDK Reference Guide orders all methods according to the Managed Object types they act upon, rather than grouping them all under the service instance. This makes it easy to see what is possible for each type of managed object and to identify what the type of the first argument to each service instance call should be.

The diagram below shows the structure of the ServiceInstance and ServiceContent managed objects. Note that the ServiceContent object contains properties representing the inventory and all the major management interfaces. Also, the rootFolder property of the service instance holds the inventory for all the VI objects (VMs, hosts, datastores, networks, etc).



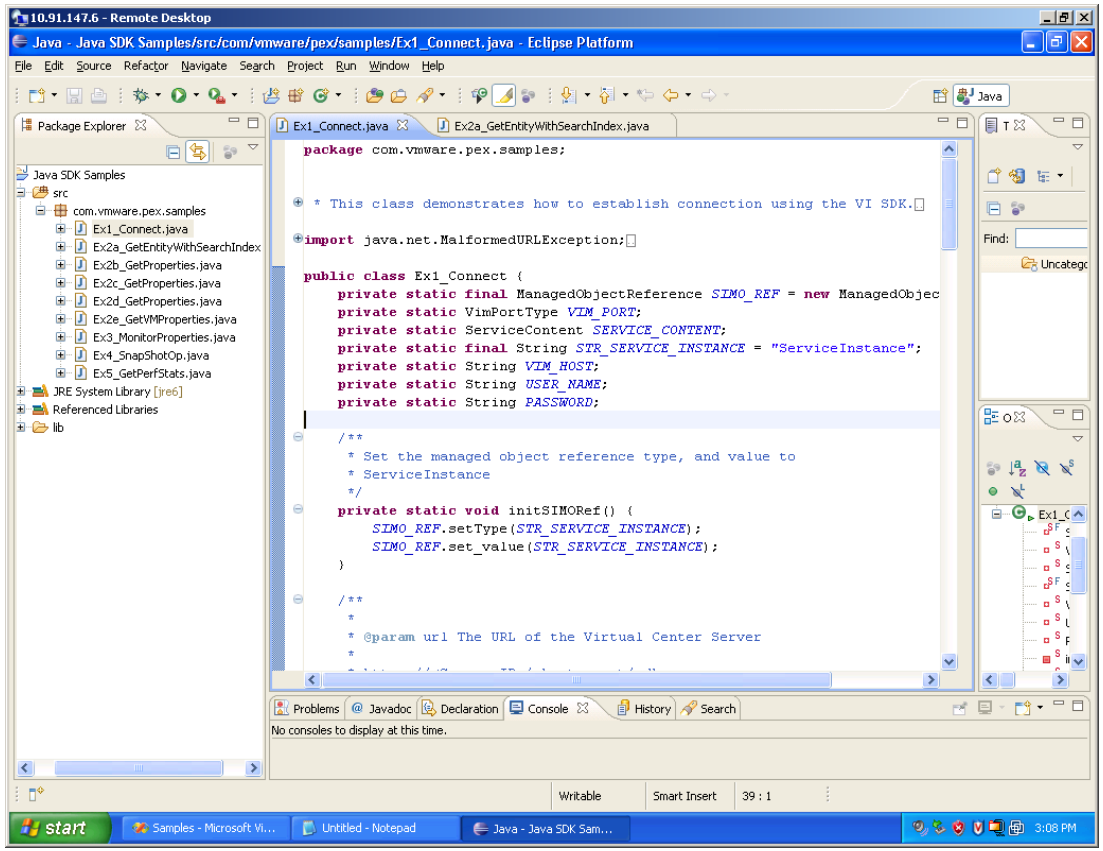
Step 2: Open the lab files for Ex1_Connect.java

2.1 If you have not already opened eclipse, begin by opening “Java Lab” on the desktop.

Your IDE should be setup for you as shown below.

2.2 The source code is set for you on the left window in the Package Explorer under *src / com.vmware.pex.samples*. Double click the icon on the left-hand panel that corresponds to the code for this exercise number (the solutions for each exercises are also listed). Just selecting it may not work as it will only change your working file if you already have it open.

2.3 You should browse the code in the main window. Please do not compile or run the code until you get to that section of the instructions.



Step 3: Review the Java features in Connect.java

This section covers some of the Java-specific code in Connect.java for folks with little or no Java experience. Experienced users should skip to the functional walkthrough, which explains what sorts of VI API interaction is occurring in each section of code.
Please look through the code you opened in the browser in the previous step.
The program begins with several import statements.

```
import java.util.*;
import java.net.*;
import com.vmware.vim25.*;
import java.text.*;
```

These statements import a few key Java libraries that are used in this lab. As described above, in the "Libraries in the lab" section, the java.util and java.text libraries contain some basic utilities, the java.net library contains networking utilities, and the com.vmware.vim25 library contains VI SDK-related utilities.

Next, the program defines the Connect class and some of the variables used in that class. The Connect class is public because it is intended to be accessed from outside the class. A number of variables are defined, but they are private, meaning they are not intended to be accessed outside the class.

```
public class Ex1_Connect {
   private static final ManagedObjectReference SIMO_REF = new ManagedObjectReference();
   private static VimPortType VIM_PORT;
   private static ServiceContent SERVICE_CONTENT;
   private static final String STR_SERVICE_INSTANCE = "ServiceInstance";
   private static String VIM_HOST;
   private static String USER_NAME;
   private static String PASSWORD;
```

The rest of the code consists of a number of method definitions that are used in the program. The next section describes their functionality in detail, but there are no specific Java features.

The part of the code that will be executed is in a special method called main( ). The main function takes some arguments and stores them in an array called args. These arguments are taken directly from the command line at execution time when a Java program is run.

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

The next line sets some system properties relating to how the Axis web toolkit processes SSL connections. In this case, the certificate processing is set to a value that results in all certificates being trusted automatically, regardless of the trust relationship. This is done in the lab environment to make the setup easier and the lab more portable. It is also sometimes useful in test/dev environments. Because the client's user credentials are going to be sent over the SSL connection, **automatically accepting the SSL certificates is a bad idea for programs that will be deployed in enterprise environments**. Avoiding proper SSL trust relationships can make it easier for man-in-the-middle attacks to compromise the connection security and read the passwords and traffic.

```
//This is to accept all SSL certifcates by default.
System.setProperty("org.apache.axis.components.net.SecureSocketFactory","org.apache.axis.components.net.SunFakeTrustSocketFactory");
```

The right way to do this is to either write your own certificate store or add the VC certificates to the certificate store managed by the OS. This is described in the VI SDK Developer Setup Guide.

The first few lines process the command line arguments. In this case, a usage statement is printed if the wrong number of arguments is used. If the correct number of arguments is used, three variables are set with the three arguments from the command line. A few other Java features are used as well. args.length represents a built-in Java value for the length of an array. The try statement is described above and is used to handle code that may potentially result in unplanned errors.

```
if (args.length < 3) {
   printUsage();
} else {
   try {
      VIM_HOST = args[0];
      USER_NAME = args[1];
      PASSWORD = args[2];
```

Step 4: Functional Walk-through of Connect.java

The **main** function of Connect.java calls several functions. Some of these are defined in the program itself, and others are calls to the VI API web service. The basic structure of the logic of the program is shown below in outline form.

- initAll ()

initAll is a parent function for various functions that initialize the environment. All the exercises in this lab will use the same parent function to initialize. However, the subfunctions may change from exercise to exercise depending on what needs to be initialized for a particular program.

- initSIMORef ()
  initSIMORef simply initializes the SIMO_Ref variable. This variable is a Managed Obejct Reference (MORef) of type "ServiceInstance" with a value of "ServiceInstance". Because the ServiceInstance is the object used to map all other MORefs to objects, its MORef is special. Instead of a MORef that looks like an object ID, the Service Instance MORef simply identifies its purpose.
- initVimPort ()
  initVimPort takes the URL of a VC or ESX sdk service as an argument (this URL is obtained indirectly from the command line options the program was run with), connects to the http port, and returns the port to use for future transactions. This does not login to the web service. There is a very important distinction here: at this point the program is connected to an http session, but has not yet connected to a webservice session. Making the distinction between these two types of connections can be critical for debugging issues with VI API connections.
  - locator.setMaintainSession(true);
    This line calls a method of the locator object. The line setting the setMaintainSession is very important. Unless this line is present, each call will need to reopen the connection to the webservice endpoint. This is equivalent to setting an http keep-alive for the connection.
  - VIM_PORT = locator.getVimPort(new URL(url));
    This line calls a method of the locator object that opens an http connection. For users unfamiliar with Java, the "new URL(url)" portion simply puts the string "url" in an object of the type "URL", which is what the locator expects.
- initServiceContent ()
  initServiceContent retrives the service content across the http connection (again, note this is not yet a webservice connection). The service content is essentially a directory of what services are available for connecting to the web service. Retrieving the service content is the **only** method that can be called without authenticating.
  - SERVICE_CONTENT = VIM_PORT.retrieveServiceContent(SIMO_REF);
    This line calls a method of the VIM_PORT object (which is of type VimPortType). The method called handles the retrieval of the Service content. This retrieval essentially grabs the WSDL file describing the web service.
- **connect ()**

This is a function you will be writing or pasting for the lab. It is currently blank and the code is covered below.
connect authenticates to the web service and connects to the web service. Note that the earlier steps were merely connected to an http port in order to get information about the webservice. The connection to the web service occurs over the http connection that has been previously setup.

- VIM_PORT.login(SERVICE_CONTENT.getSessionManager(), uname, pword, null);
  This line calls another method of the VIM_PORT object (which is of type VimPortType). The method called here handles the login. Note that it uses the session manager object obtained in the previous type and accessed by calling the getSessionManager method on the SERVICE_CONTENT object obtained in the previous step. The uname and pword variables specify the username and password to use. These are obtained indirectly from the command line options when the program was invoked.
  Another useful note here is that the last argument to the login method describes the locale to use. All strings returned by VC are formatted for this locale (if it is available). Using null here indicates the user will use the default locale that VC is set for. This may be confusing if VC is set for a different locale than the program.

- **getCurrentTime();**
  This is a function you will be writing or pasting for the lab. It is currently blank and the code is covered below.
  getCurrentTime simply gets the current time on the VC server.
  - VIM_PORT.currentTime(SIMO_REF)
    This line calls the currentTime method of the VIM_PORT object. VIM_PORT is an object of type VimPortType.
- PrintServerTime()
  PrintServerTime simply formats and prints the time returned by the previous function in a human-readable form. No VI SDK methods are used for this.
- disconnect()
  disconnect handles the disconnection from the server. Remembering to disconnect (both for successful and unsuccessful executions) is an important part of any VI SDK program. By default, VC can have 100 web service connections open at a time. The Java Axis toolkit will by default automatically timeout an inactive socket after 10 minutes. VC and ESX timeout sessions after 30 minutes of inactivity (except sessions in the middle of a call to waitForUpdates( )).

Step 5: Modify the Code

For this lab, you'll be making changes to 2 functions.

The first function you'll modify makes the http connection to the port the webservice runs on (without actually connecting to the web service). This method takes the url of the sdk service on a VC host as an argument and performs the http connection. The first step is to create the VimServiceLocator, then to use the locator object to get the VimPort.

Another critical line of code here requests that the http connection be kept alive (httpkeepalive) by setting the locator's setMaintainSession property to TRUE.

5.1 Fill in the initVimPort function as follows:

```
private static void initVimPort(String url) {
        VimServiceLocator locator = new VimServiceLocator();
        locator.setMaintainSession(true);
        try {
            VIM_PORT = locator.getVimPort(new URL(url));
        } catch (MalformedURLException mue) {
            mue.printStackTrace();
        } catch (Exception se) {
            se.printStackTrace();
         }
}
```

The second function performs the connection to the webservice. Keep in mind this is after an http connection has already been opened and the service content has been located, as described in the walk-through above. The code you'll be adding here will need to use the SessionManager's login method to connect to the web service. As with all method calls on managed objects, the method actually belongs to the service instance (VIM_PORT). In order to get the MORef of the service instance, you can use the getSessionManager method of the Service content.

Note: the url parameter is unused. It can be used if you'd like to make some modifications and perform more of the connection process in a single function.

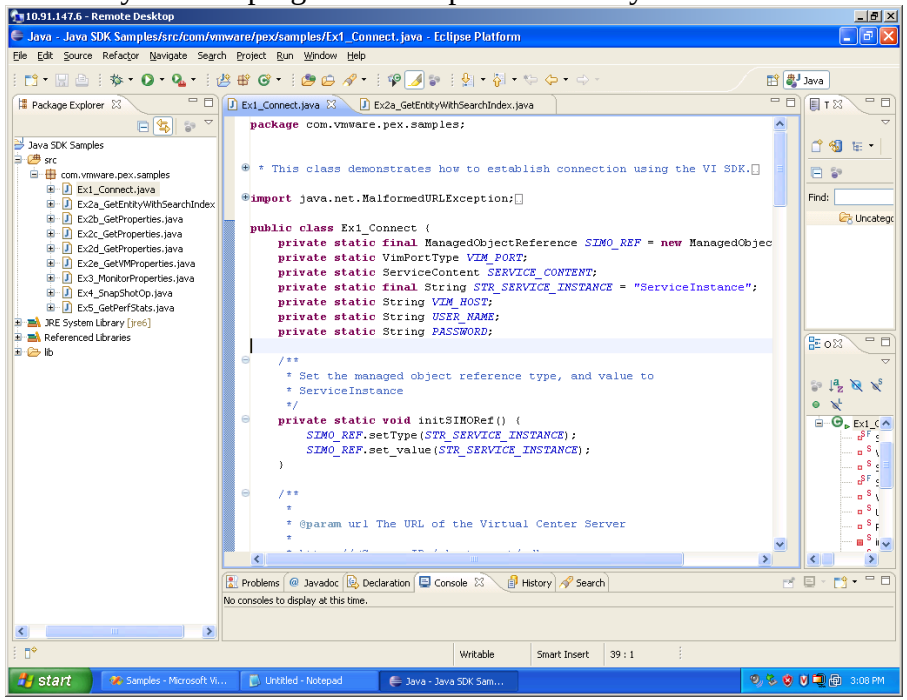
5.2 Fill in the connect () function as follows:

```
private static void connect(String url, String uname, String pword) throws Exception {
        VIM_PORT.login(SERVICE_CONTENT.getSessionManager(), uname, pword, null);
    }
```

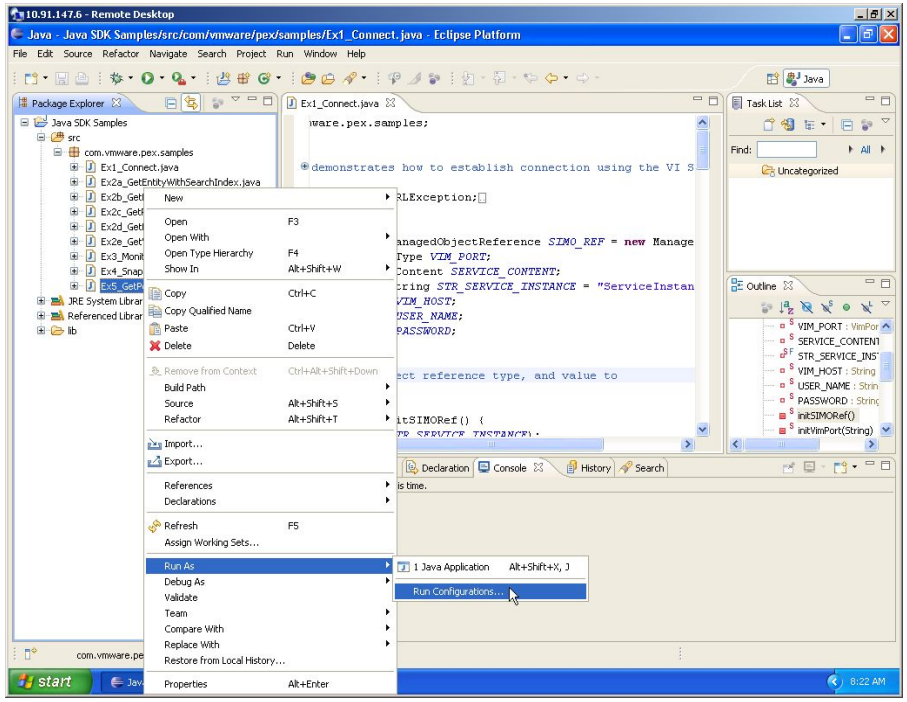
Step 6: Compiling and Running the Code

Once you've made the necessary changes to the code, compile and run the code. Despite your normal preferences when running and compiling code, please use the procedure in the lab to make it as easy as possible for the lab staff to diagnose any problems you have.

6.1 Verify that the program is complete and that you have filled in all the code sections specified in step 5.



6.2 Right click on the file that you were working on in the Package Explorer, go to “Run As” and click “Run Configurations”. A new window will open up with a space for providing the arguments to the command. Recall that a number of the variables in the program are taken from the command line options.

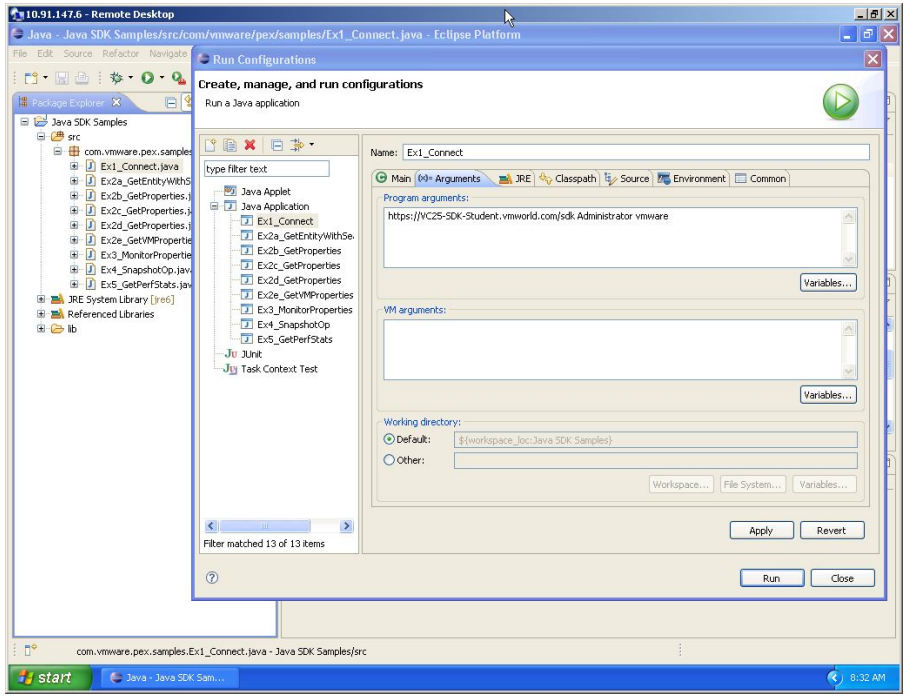


6.3 On the right (Run Configuration) window, confirm up top that it's the correct file and click on the "Arguments" tab. This will also be the file that is highlighted in the pane on the left side of the Run Configuration window. If you do not have the correct file selected, you will re-run the last program.

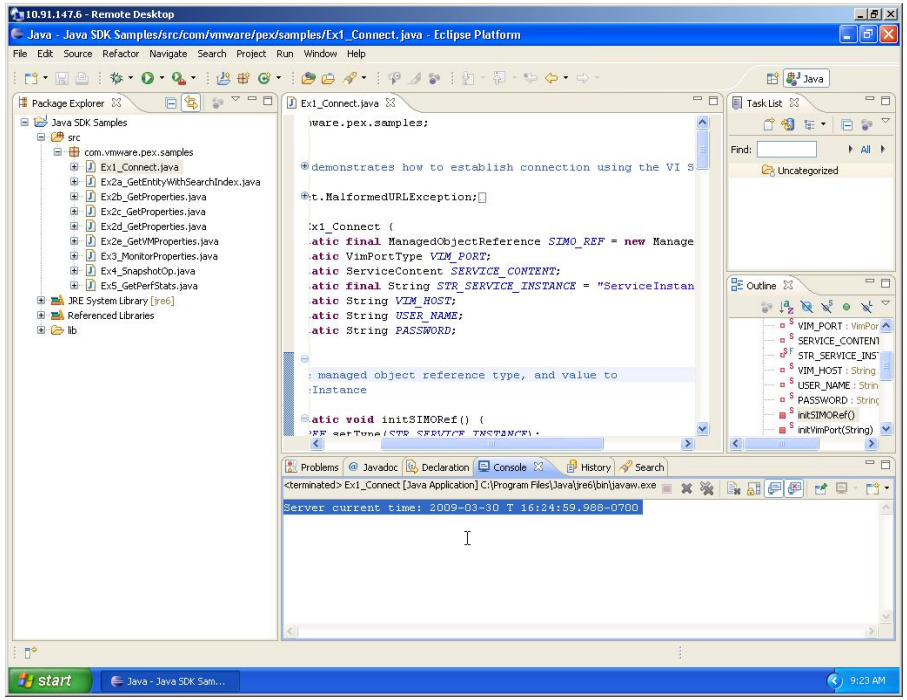
Input the arguments that that this program requires. You will need to input the following:

1) Virtual Center host (the VIM_HOST listed at your station) - *this is the Virtual Center hostname or IP*
2) Username (the USERNAME listed at your station) – *the account that you will be authenticating with*
3) Password (the PASSWORD listed at your station) – *the account's password*

Usage: https://<VIM_HOST>/sdk <USERNAME> <PASSWORD>
Example : https://vc01-student01.vmworld.com/sdk sdk-student01 abc123



6.4 Click Apply and Run Application at the bottom of the screen. You should see the correct output in the Console tab just below your code.



Hands-on Exercise 2a: Getting Properties of an Object

Exercise 2 teaches how to find managed objects in the VMware Infrastructure and how to list the properties of those objects. The basic concepts of finding managed objects and listing their properties are applicable to all objects in VMware Infrastructure, including virtual machines, hosts, virtual switches, port groups, resource pools, datastores, and more. Using these methods, you will be able to locate any of the configuration or state information for any object. Note that performance information is not considered a basic property of a VMware Infrastructure object. Performance metrics will be covered in Exercise 5.

This exercise is split into several parts. Part 2a covers how to view the properties of an object you already have the MORef of. It uses a special object called the SearchIndex which can find an object's MORef based on specific criteria. Part 2b explains how to get the MORef of an object if you don't already know it or can't get it via the SearchIndex. Parts 2c and 2b explain variations on finding objects, and part 2e describes some of the important performance issues around locating objects.

Step 1: Understanding Managed Objects and Properties

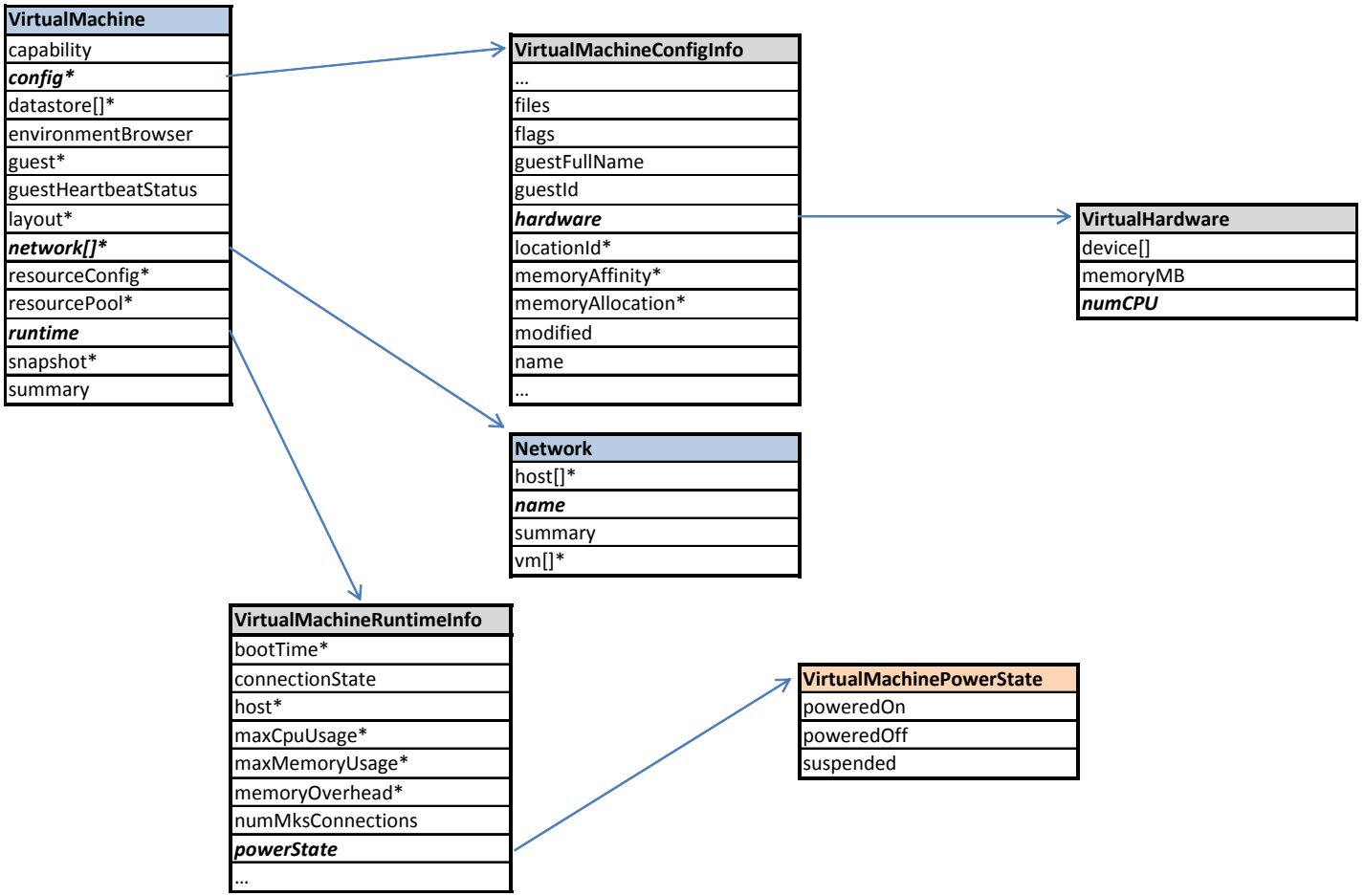
The goal of this exercise is to gain a basic understanding of the Managed objects used for programming the VMware Infrastructure. These objects generally map to the objects that are visible in the VMware Infrastructure Client (VI Client) GUI. The following are examples of the types of managed objects that typical VI SDK programs can manipulate:

- ComputeResource
- Datacenter
- Datastore
- Folder
- HostSystem
- Network
- ResourcePool
- Task
- VirtualMachine

Each of these managed objects has several associated properties. These properties represent both the configuration and the state of the object. In addition, these properties can indicate relationships with other objects. For example, the objects stored in a folder would be stored in its `childEntity` array property.

The following diagram of a VirtualMachine object illustrates some of the properties that you will use in this exercise. Note that any field name with square brackets following it may have multiple instances and must be handled as an array. Fields used for these exercises are highlighted in bold italics, blue object headers are managed objects, gray represents data objects, and orange is for enumerated-type values.

This diagram is by no means a complete list of properties. You can find a complete list of properties for each object in the *VI SDK Reference Guide* (http://www.vmware.com/support/developer/vc-sdk/visdk25pubs/ReferenceGuide/index.html). You can also use the Managed Object Browser (MOB) tool to see the relationships and properties of server-side objects on a live system.



The first step to interacting with objects in a VI SDK program is specifying the objects. You can do so by using a local reference for the actual server-side object; this reference is called a *managed object reference.*

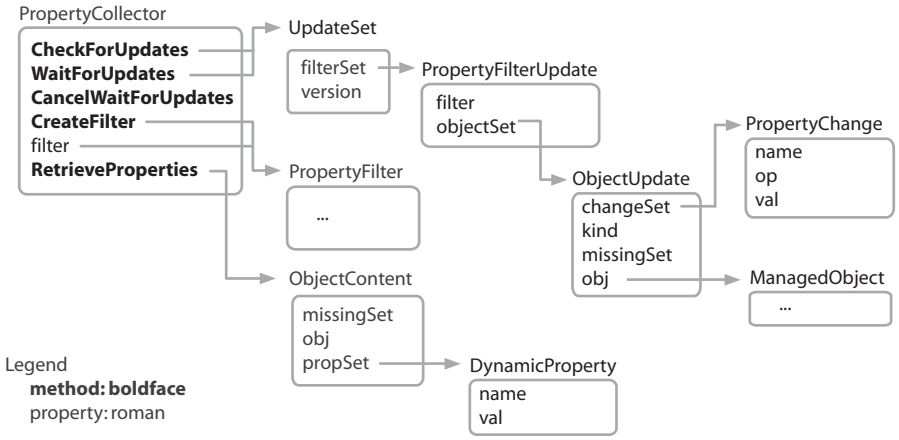
The second step to interacting with objects is to determine the properties that you want to examine. The previous diagram shows some properties of a VirtualMachine server-side object. As mentioned earlier, you can use the *VI SDK Reference Guide* to find this information for other objects. Although you cannot manipulate the managed object directly, you can access the properties using the property collector. Once you have collected properties with the property collector, the resulting objects are data objects. You can manipulate data objects locally using normal Java syntax (without requiring additional property collector calls, unless the objects have updated).

The PropertyCollector

A PropertyCollector is a managed object type that stores a number of PropertyFilters that clients use to retrieve a set of properties from one or more managed objects (VMs, hosts, datacenters, etc). Because an object's children (for instance VMs in a folder or folders in a datacenter) are also properties, the property collector is an efficient way of storing ways to find managed objects with specific attributes. **All PropertyCollector state information is session-specific. A client cannot share its property collector filters with other clients or use them after a session terminates.**

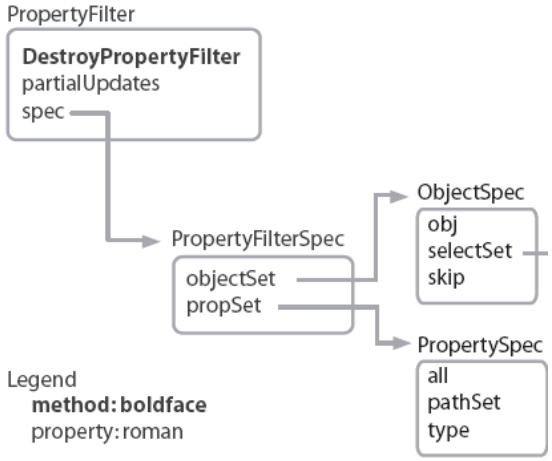
Side-note: Clients can also use the PropertyCollector to determine when any of the properties they are interested in have changed. The change detection mechanism supports both polling (using CheckForUpdates method) and notification (using WaitForUpdates method) to detect property changes. To decrease network traffic when collecting updates, a client creates one or more property collector filters to specify the parts of a managed object in which the client is interested. This method is used in Exercises 3 and 4.

The diagram below shows the data structure of the PropertyCollector.



PropertyCollector is essentially the management point for PropertyFilters. A PropertyFilter is a specification of properties to filter on (stored in a PropertyFilterSpec), along with management interfaces for deleting the filter and tracking changes to the filtered properties. Since properties include managed objects, this is basically a filter for any managed objects or properties you are interested in.

1.1 Lookup the PropertyFilter and PropertyCollector in the SDK Reference Guide. The Diagram below shows the objects making up a PropertyFilter:



In order to create a PropertyFilter, you will need to create a PropertyFilterSpec, which contains an ObjectSpec (representing how to execute a search) and a PropertySpec (representing what objects to filter and what properties to filter in them). The PropertyFilterSpec is the data object that transfers the data about the properties to be returned to the propertyfilter spec (which is a managed object and thus can't be manipulated directly).

The ObjectSpec part of a PropertyFilterSpec

1.2 Look ObjectSpec up in the SDK Reference Guide (it's a data object). As shown in the figure above and the SDK Reference Guide, the properties of an ObjectSpec are:

- *obj* (A ManagedObjectReference pointing to the root of the filter)

- *selectSet* (the optional array of SelectionSpec objects specifying how to traverse the hierarchy looking for addition objects to filter)
- *skip* (a Boolean specifying whether or not to report the properties of the root object).

Note that all the properties here revolve around how to navigate the inventory tree, rather than object selection directly.

For this exercise, *obj*, the ManagedObjectReference, is going to be the ManagedObjectReference returned by the SearchIndex method used above, since this exercise jumps directly there to search for properties. The *SelectSet* property will be empty, since there is no need to traverse the inventory tree to find the appropriate objects. The *skip* property will be false, since the root object is the object that will have the properties being searched for.

The PropertySpec part of a PropertyFilterSpec

It may also be helpful to lookup the PropertySpec in the SDK Reference Guide. As shown in the figure above and in the SDK Reference Guide, the properties of a PropertySpec are:

- *all* (a boolean indicating whether you want all the properties of the object or only those in pathSet)
- *pathSet* (an optional array of strings representing object properties you want to filter)
- *type* (a string indicating the managed object type that you will be filtering—what you care about).

For the program, it will be necessary to determine what values these properties should have. The *all* property will be false, since only specific properties will be selected. The type will be “VirtualMachine” (this is a string and thus includes the quotes), since you are trying to output VM properties. However, outside of the lab, the type here could be any ManagedObject name. This list of valid ManagedObject types is on the ManagedObjectReference data object page in the reference guide.

1.3 Explore the VirtualMachine object to determine what properties contain the information you are looking for. When *pathSet* is initialized, the goal is to create a String array. One element will be the VM name. This is necessary because the properties are returned in an unordered list, and it is important to key off something to make certain you are collecting properties for the right VM. Note that the VM name property is not in the Reference Guide as a VirtualMachine property because this property is inherited from ManagedObject (and a VirtualMachine is a ManagedObject). The other properties used in this example happen to be under the *config* property, so setting a pathSet will look something like this:

```
ps1.pathSet = new String[] { "name", "config.<property>", "config.<property>", "config.<property>" };
```

In the above example, the code only specifies specific sub-properties of *config*, rather than the entire *config* object. This lessens the impact on the server and increases response times, since all the data must be generated on the server and then sent over the wire wrapped in XML.

1.4 Look up the SearchIndex Object in the Reference Guide. In the Reference Guide, click on “Managed Object Types” and type “search” in the search field. Select SearchIndex, the only option, to get a description of what it can do. The SearchIndex is a shortcut for searching for objects using the most common properties. It is important that the properties searched for be unique, as the SearchIndex API will only return one object.

Exercise 2b shows how to do non-standard searches that are not based off pre-indexed values.

The next step is to find the appropriate methods of the SearchIndex object. Verify the arguments needed by this object in the reference guide. Note the types of the arguments, which you will use when you call the method in your code. In Java and C#, strings should be surrounded by double quotes, Booleans either have the value true or false, and you can leave an argument blank by filling it with null.

Alternative: because the SearchIndex is a managed object, it can be found in the MOB. The SearchIndex managed object is located off the content link when you open the MOB. Click the appropriate method to open a window for entering arguments, and then invoke the method and see the results by clicking the appropriate link. There is no need to specify a Data Center, and the program will be searching for the VM with the IP address appropriate for your lab (this will given to you by the instructor). An invocation result of void means you either used incorrect syntax or used an incorrect IP address. One other thing to consider is that VM IP addresses are gathered from VMware Tools, so they are only available to be searched if the VM is on and tools is running.

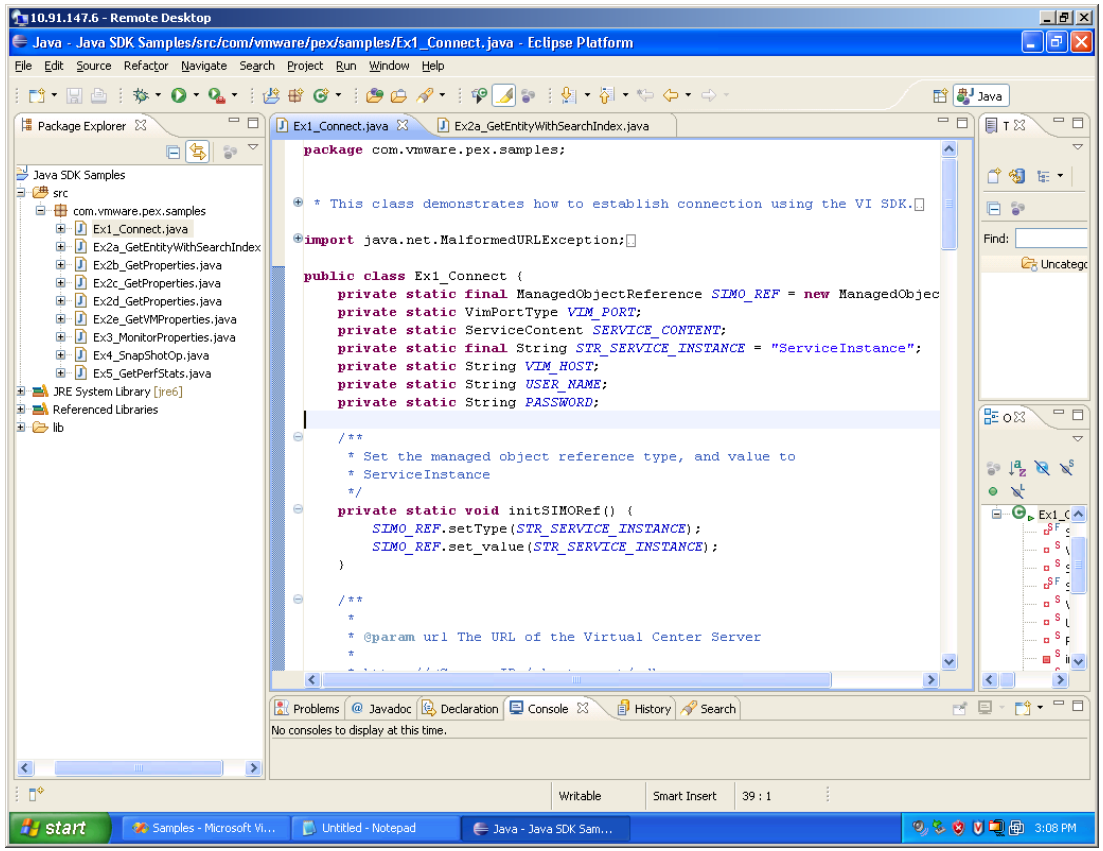
Step 2: Open the lab files for GetEntityWithSearchIndex.java

2.1 If you have not already opened eclipse, begin by opening “Java Lab” on the desktop.

Your IDE should be setup for you as shown below.

2.2 The source code is set for you on the left window in the Package Explorer under *src / com.vmware.pex.samples*. Double click the icon on the left-hand panel that corresponds to the code for this exercise number (the solutions for each exercises are also listed). Just selecting it may not work as it will only change your working file if you already have it open.

2.3 You should browse the code in the main window. Please do not compile or run the code until you get to that section of the instructions.



Step 3: Functional Walk-through of GetEntityWithSearchIndex.java

The **main** function of Connect.java calls several functions. Some of these are defined in the program itself, and others are calls to the VI API web service. The basic structure of the logic of the program is shown below in outline form.

- initAll ( )
  - initSIMORef ()

    This function was described in the previous exercise

- initVimPort ()

  This function was described in the previous exercise
- initServiceContent ()

  This function was described in the previous exercise
- connect ()

  This function was described in the previous exercise
- initPropertyCollector ()
  - PROP_COLLECTOR = SERVICE_CONTENT.getPropertyCollector();
- initRootFolder ()
  - ROOT_FOLDER = SERVICE_CONTENT.getRootFolder();
- initSearchIndex ()
  - SEARCH_INDEX = SERVICE_CONTENT.getSearchIndex();

- retVal = getManagedEntity(isVM, searchType, searchParam);

  getManagedEntity is a generic function that uses the SearchIndex to find the MORefs for VMs and hosts based on a variety of criteria (datastore path, DNS name, inventory path, IP address, or UUID). The function is simply an "if" statement separating Searches on VMs from searches on hosts, and a series of if statements separating the searches for different types of criteria.
  - retVal = VIM_PORT.findByDatastorePath(SERVICE_CONTENT.getSearchIndex(), null, searchParam);

    This line calls the findByDatastorePath method, which takes a datastore path specified in an argument and returns the MORef of the VM that has that path. As with all method calls related to managed objects, the method is actually a method of the service instance (VIM_PORT), and the managed object being referred to is listed as the first argument. In this case, the method is associated with the SearchIndex, so the MORef of the SearchIndex is the first argument. The second argument indicates the specific datacenter to search; null indicates that all datacenters will be searched. The third argument is the search parameter (this comes indirectly from the command line when executing the program), in this case the datastore path to the VM's vmx file. Because only VMs can be searched for by datastore path, there is no fourth argument indicating whether the search is for a VM or host.
  - retVal = VIM_PORT.findByDnsName(SERVICE_CONTENT.getSearchIndex(), null, searchParam, true);

This line calls the findByDnsName method, which returns the first VM it finds with the specified DNS name (two or more VMs could be configured with the same DNS name and there is no guarantee which will be returned). This is otherwise similar to the format of the previous function, with the exception that a fourth argument is included which specifies that the object being searched for is a VM. If the fourth argument is false, the search would be for an ESX host.

- retVal = VIM_PORT.findByInventoryPath(SERVICE_CONTENT.getSearchIndex(), searchParam);

  This line calls the findByInventoryPath method, which returns the the VM or host it finds with the specified inventory path. This is otherwise similar to the format of the previous function. There is no fourth argument, since the inventory path is always enough information to distinguish a VM from an ESX host.

- retVal = VIM_PORT.findByIp(SERVICE_CONTENT.getSearchIndex(), null, searchParam, true);

  This line calls the findByIp method, which returns the first VM it finds with the specified IP address (two or more VMs could be configured with the same IP address and there is no guarantee which will be returned). This is otherwise similar to the format of the findByDnsName function.

- retVal = VIM_PORT.findByUuid(SERVICE_CONTENT.getSearchIndex(), null, searchParam, true, true);

  This line calls the findByUuid method, which returns the first VM it finds with the specified Uuid (although the inventory UUID is unique, two or more VMs could be configured with the same BIOS UUID and there is no guarantee of which will be returned). This is otherwise similar to the format of the findByIp function, except that there is also a final argument that specifies whether to search for a VM instance UUID (true) or BIOS UUID (false).

- retVal = VIM_PORT.findByDnsName(SERVICE_CONTENT.getSearchIndex(), null, searchParam, false);

  This line calls the findByDnsName method, which returns the first ESX host it finds with the specified DNS name (two or more hosts could be configured with the same DNS name and there is no guarantee which will be returned). The format and functionality of this call is the same as described above for VMs, but the last argument is false to indicate this acts on ESX hosts.

- retVal = VIM_PORT.findByInventoryPath(SERVICE_CONTENT.getSearchIndex(), searchParam);

  This line calls the findByInventoryPath method, which returns the the VM or host it finds with the specified inventory path. This is otherwise similar to the format of the previous function. There is no fourth argument, since the inventory path is always enough information to distinguish a VM from an ESX host.

- retVal = VIM_PORT.findByIp(SERVICE_CONTENT.getSearchIndex(), null, searchParam, false);

  This line calls the findByIp method, which returns the first ESX host it finds with the specified IP address (two or more hosts could be configured with the same IP address and there is no guarantee which will be

returned). The format and functionality of this call is the same as described above for VMs, but the last argument is false to indicate this acts on ESX hosts.

- retVal = VIM_PORT.findByUuid(SERVICE_CONTENT.getSearchIndex(), null, searchParam, false, false);

  This line calls the findByUuid method, which returns the first host it finds with the specified Uuid. The format and functionality of this call is the same as described above for VMs, but the fourth argument is false to indicate ESX hosts will be searched. The fifth argument is always false for hosts, since there is no separate instance and BIOS UUID for hosts.

- **getVMInfo( );**

  This is a function you will be writing or pasting for the lab. It is currently blank and the code is covered below. getVMInfo uses the MORef obtained in the previous step to look up the properties of the managed object the MORef refers to. The first part of this function builds the data structure for a PropertyFilterSpec and the PropertySpec and ObjectSpec that make it up. This is critical to understand and is described in detail in the introductory section of this exercise.

  - ObjectContent[] oCont = VIM_PORT.retrieveProperties(PROP_COLLECTOR, propertyFilterSpecs);

    This line uses the retrieveProperties method to get the properties (specified in the PropertySpec) from the managed object (retrieved in the searchIndex calls above and encapsulated in the ObjectSpec). Note that like all methods related to managed objects, the method belongs to the service instance and the managed object it relates to is the first argument.

- disconnect();

  The method disconnects the webservice and described in the sections above.

Step 4: Modify the Code

The code you'll be filling in for this lab is a part of the getVMInfo function. This code builds the propertyFilterSpec that defines what properties of which managed object should be returned to the client.

In this case, the managed object reference is provided as an argument (having been returned by a searchIndex request earlier in the program). You'll supply a propertyFilterSpec, and then the rest of the code puts the retrieved data into a convinient string format.

The propertyFilterSpec you build should get the name and the power state of a virtualMachine.

In this case, you know the object, so the object spec just points to that.

objectSpec

Obj = mor (a managed object ref from the argument)

The properties you need can be looked up in the VirtualMachine section of the VI SDK Reference Guide (keep in mind that name is inherited from ManagedObject).

propertySpec

All = FALSE (only get requested properties, not all properties)

PathSet = { "name", "runtime.powerState" } (the names of the properties to get)

Type = "VirtualMachine" (the type of managed object being examined)

The PropertyFilterSpec defines all the objectsSpecs to get (an array) and all the properties to get (an array). Here you'll just be getting the one element in each array. Using multiple elements in the array requires some concepts covered later in the lab.

PropertyFilterSpec

PropSet [ ]

propertySpec All = FALSE PathSet = { "name", "runtime.powerState" } Type = "VirtualMachine"	…	…	…

ObjectSet

objectSpec Obj = mor	…	…	…

4.1 Fill in the getVMInfo () function as follows:

```
  public static void getVMInfo(ManagedObjectReference mor) {
     try {
// Begin fill-in
// v v v v v v v
        // Create Property Spec
        PropertySpec propertySpec = new PropertySpec();
        propertySpec.setAll(Boolean.FALSE);
        propertySpec.setPathSet(new String[] { "name", "runtime.powerState" });
        propertySpec.setType("VirtualMachine");
        PropertySpec[] propertySpecs = new PropertySpec[] { propertySpec };

        // Now create Object Spec
        ObjectSpec objectSpec = new ObjectSpec();
        objectSpec.setObj(mor);
        ObjectSpec[] objectSpecs = new ObjectSpec[] { objectSpec };

        // Create PropertyFilterSpec using the PropertySpec and ObjectPec
        // created above.
        PropertyFilterSpec propertyFilterSpec = new PropertyFilterSpec();
        propertyFilterSpec.setPropSet(propertySpecs);
        propertyFilterSpec.setObjectSet(objectSpecs);

        PropertyFilterSpec[] propertyFilterSpecs = new PropertyFilterSpec[] { propertyFilterSpec };

        ObjectContent[] oCont = VIM_PORT.retrieveProperties(PROP_COLLECTOR,
              propertyFilterSpecs);
// ^ ^ ^ ^ ^ ^
// End fill-in

        if (oCont != null) {
           // System.out.println("ObjectContent Length : " + oCont.length);
           StringBuilder sb = new StringBuilder();
           for (ObjectContent oc : oCont) {
              DynamicProperty[] dps = oc.getPropSet();
              if (dps != null) {
                 for (DynamicProperty dp : dps) {
                    if (dp.getName().equalsIgnoreCase("name")) {
                       sb.append(dp.getVal());
                       sb.append(" : ");
                    } else {
                       sb.append(dp.getVal());
```

```
                        sb.append("\n");
                    }
                    // System.out.println(dp.getName() + " : " +
                    // dp.getVal());
                }
            }
        }
        System.out.println(sb.toString());
      }
    } catch (Exception e) {
       e.printStackTrace();
    }
 }
```

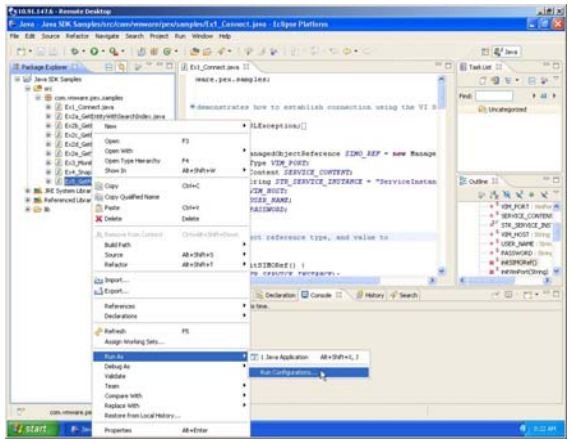
Variation (optional)

Look through the virtualMachine object in the Reference Guide. Try replacing the properties in the code with alternative properties. Also, you can use different command line options and explore the hostSystem object.

Step 5: Compiling and Running the Code

Once you've made the necessary changes to the code, compile and run the code. Despite your normal preferences when running and compiling code, please use the procedure in the lab to make it as easy as possible for the lab staff to diagnose any problems you have.
5.1 Verify that the program is complete and that you have filled in all the code sections specified in step 5.

5.2 Right click on the file that you were working on in the Package Explorer, go to “Run As” and click “Run Configurations”. A new window will open up with a space for providing the arguments to the command. Recall that a number of the variables in the program are taken from the command line options.

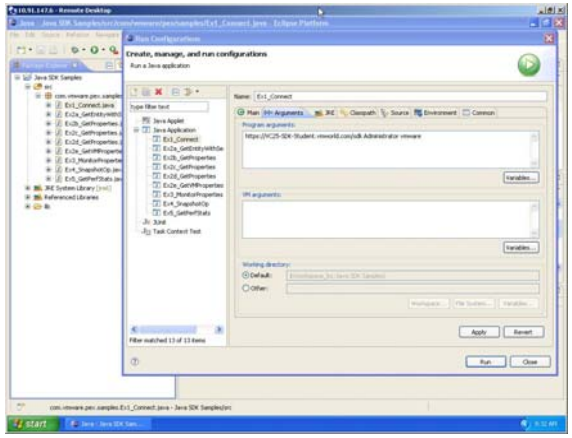


5.3 On the right (Run Configuration) window, confirm up top that it's the correct file and click on the "Arguments" tab. This will also be the file that is highlighted in the pane on the left side of the Run Configuration window. If you do not have the correct file selected, you will re-run the last program.

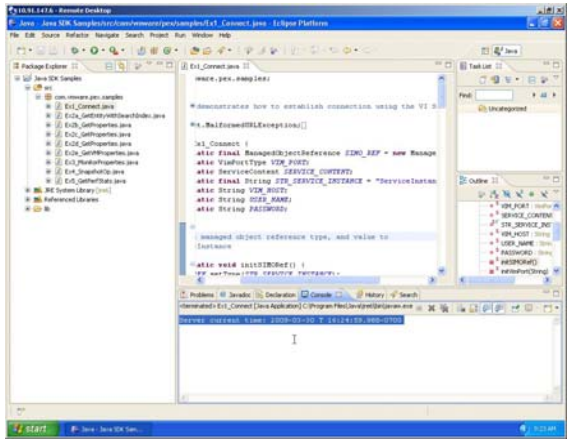
Input the arguments that that this program requires. You will need to input the following:

1) Virtual Center host (the VIM_HOST listed at your station)
2) Username (the USERNAME listed at your station)
3) Password (the PASSWORD listed at your station)
4) TRUE –Indicates TRUE if the search is for a VM
5) Entity IP (the MY_VM_IP listed at your station) – *what entity of the VM are you using as an identifier?*

Usage: https://<VIM_HOST>/sdk <USERNAME> <PASSWORD> TRUE ip <MY_VM_IP>
Example : https://vc01-student01.vmworld.com/sdk sdk-student01 abc123 TRUE ip 10.2.1.4



5.4 Click Apply and Run Application at the bottom of the screen. You should see the correct output in the Console tab just below your code.



Hands-on Exercise 2b: Finding an Object and Getting its properties

Exercise 2a covered how to view the properties of an object you already have the MORef of. It uses a special object called the SearchIndex which can find an objects MORef based on specific criteria. Part 2b explains how to get the MORef of an object if you don't already know it or can't get it via the SearchIndex.

Exercise 2b teaches how to find managed objects in the VMware Infrastructure and how to list the properties of those objects. The basic concepts of finding managed objects and listing their properties are applicable to all objects in VMware Infrastructure, including virtual machines, hosts, virtual switches, port groups, resource pools, datastores, and more. Using these methods, you will be able to locate any of the configuration or state information for any object. Note that performance information is not considered a basic property of a VMware Infrastructure object and requires another method for retrieval. Performance metrics will be covered in Exercise 5.

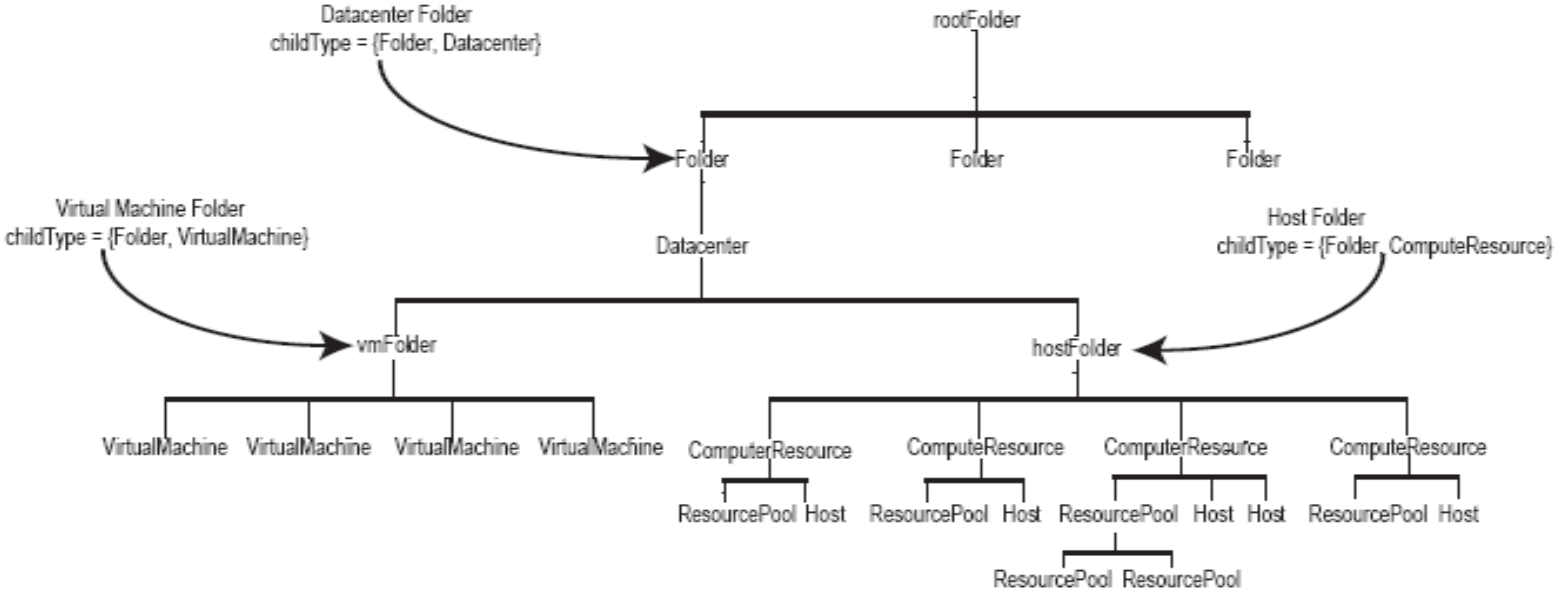
NOTE: beginning students will probably find it easiest to do exercise 2a and skip exercise 2b, which covers more complex searches. Advanced students should still read through exercise 2a in the handbook, as it describes some important concepts for this exercise.

This exercise demonstrates the use of the Property Collector to obtain a list of managed **VirtualMachine** objects by a VI Web Service Instance as well as their configuration information and then print this configuration for a specific Virtual Machine whose display name we have given as an argument to the program.

Step 1: Understand Traversal Specs and the Object Heirarchy

The Object Hierarchy

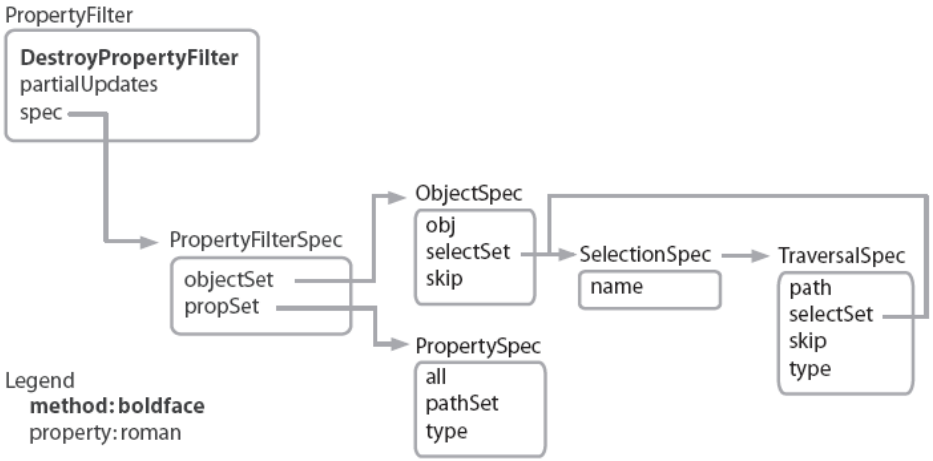
The diagram below shows a simplification of the object hierarchy. Note that folders are recursive and can contain other folders. The ESX object hierarchy is similar, but has a few differences that you should be aware of. This is discussed in detail in the Programming Guide.



The TraversalSpec

Exercise 2a showed how to locate a VM using the SearchIndex. The SearchIndex is an easy way to find managed objects using the most common properties to be searched on. This exercise will demonstrate another method that does a generalized search. In order to make the difference clear, the code is very similar to 2a.

Look up the TraversalSpec Object in the Reference Guide. As the diagram below shows, the ObjectSpec *pathSet* property can contain a SelectionSpec. A SelectionSpec describes how to traverse the object tree. The only important thing you need to know about the SelectionSpec is that it is a container with which to name a TraversalSpec. It is necessary to name a TraversalSpec in order to reuse it. One particular example of reuse is to allow recursion, for instance to allow moving down multiple levels of nested folders.



The **TraversalSpec** has four properties:

- *Path*, which is the name (a string) of any property of the currently selected object that contains arrays of managed objects. Because many ManagedObjects have multiple types of children (for instance a DataCenter has datastores, networks, host folders, and vmfolders), this lets you specify which you want to consider the child for this type of traversal.
- *SelectSet*, which allows you to point to other SelectionSpec objects to be used. Essentially this is a way to allow multiple types of traversal steps. For instance, you can traverse from a datastore to a vmfolder and then from a vmfolder to a VirtualMachine. Without this functionality, all traversals could only go one step deep.
- *Skip*, which is a Boolean allowing you to specify whether or not to filter on the objects in the path field.
- *Type*, which specifies the object type containing the properties you are searching for.

A recursive TraversalSpec

Because this exercise requires locating a VM, the inventory traversal must go from a DataCenter to a folder and then from a folder to a VirtualMachine. In addition, it is necessary to handle the case where there are multiple levels of folders, requiring a recursive call to the TraversalSpec that traverses from a folder to its child entities.

The *dc2f* TraversalSpec traverses from the DataCenter to the vmFolder. It will thus have the type of “Datacenter” (recall these types are listed in the reference guide under the ManagedObjectReference object). The MOB provides a good way to

examine the path from a datacenter to a virtual machine to determine what property should be followed here. The skip property will be false. It will be necessary to create a recursive selectionSpec with a name matching the name of a second traversal spec you will be creating called fd2c.

The second traversal spec traverses folders and their children. The type here is “Folder”. You can use the MOB to determine the appropriate path. Skip is again false, and you will be specifying two SelectionSpecs in the selection spec array—one is a recursive reference to this spec, and the other is the dc2f spec. This handles the case where there could be multiple levels of folders above or below the datacenter.

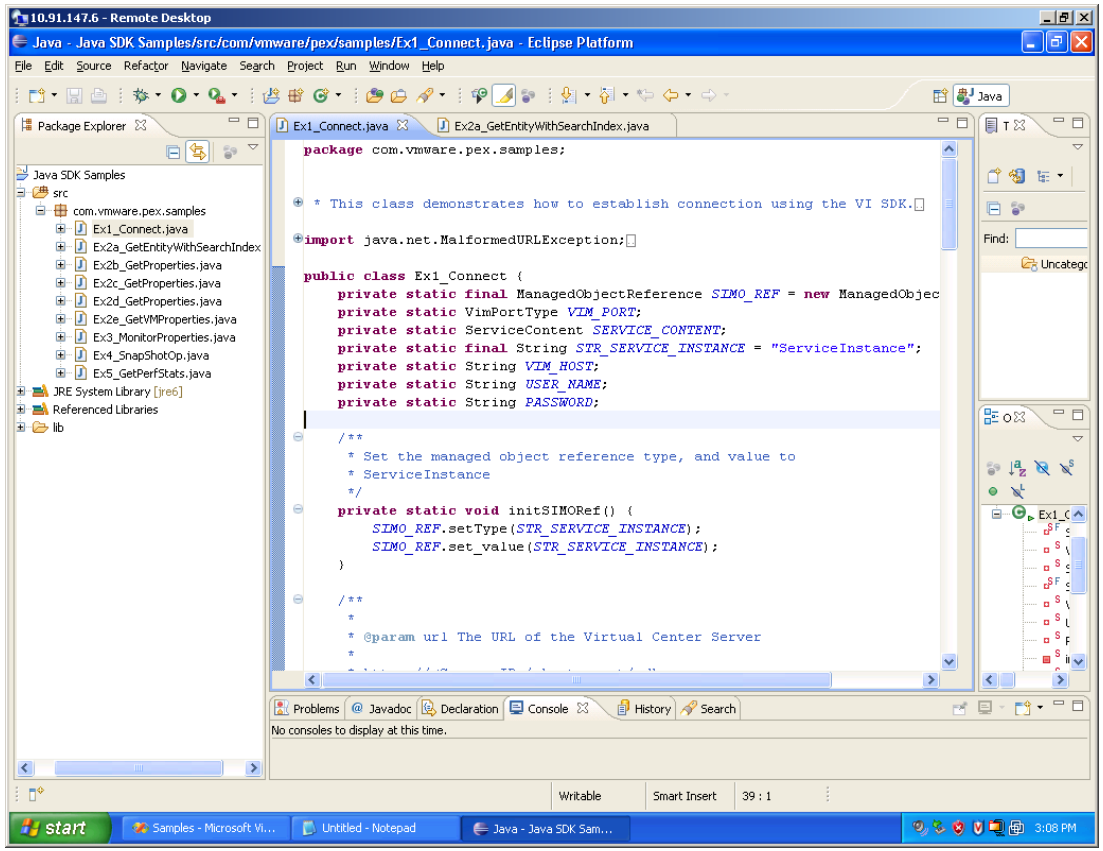
Step 2: Open the lab files for Ex2b_GetProperties.java

2.1 If you have not already opened eclipse, begin by opening “Java Lab” on the desktop.

Your IDE should be setup for you as shown below.

2.2 The source code is set for you on the left window in the Package Explorer under *src / com.vmware.pex.samples*. Double click the icon on the left-hand panel that corresponds to the code for this exercise number (the solutions for each exercises are also listed). Just selecting it may not work as it will only change your working file if you already have it open.

2.3 You should browse the code in the main window. Please do not compile or run the code until you get to that section of the instructions.



Step 3: Functional Walk-through for Ex2b_GetProperties.java

- initAll ()
  - initSIMORef ()
  - initVimPort ()
    - locator.setMaintainSession(true);
    - VIM_PORT = locator.getVimPort(new URL(url));
  - initServiceContent ()

        - SERVICE_CONTENT = VIM_PORT.retrieveServiceContent(SIMO_REF);
    - connect ()
        - VIM_PORT.login(SERVICE_CONTENT.getSessionManager(), uname, pword, null);
    - initPropertyCollector ()
        - PROP_COLLECTOR = SERVICE_CONTENT.getPropertyCollector();
    - initRootFolder
        - ROOT_FOLDER = SERVICE_CONTENT.getRootFolder();

- getVMInfo()
  getVMInfo finds all VMs and prints their powerstate. This requires using the PropertyCollector with a traversalSpec, rather than the SearchIndex used in exercise 2a, since there is no SearchIndex for finding things by VM name (only by DNS name).
  The first part of the getVMByName function calls getVMTraversalSpec to create the traversal spec.
    - **getVMTraversalSpec()**
      This is a function you will be writing or pasting for the lab. It is currently blank and the code is covered below.
      getVMTraversalSpec builds and returns a TraversalSpec that can be used later to find all VMs. Note that the TraversalSpec is just an object that describes how to traverse to find specific objects. It does not actually find the objects. Think of a Traversal spec like a map of how to find particular objects.

  The next part creates the PropertyFilterSpec. It uses the same syntax as exercise 2a, except this time the ObjectSpec has a selectionSpec that includes the TraversalSpec used to find the VM objects. After building the PropertyFilterSpec, it is used in a retrieve properties call. This is the same call used in exercise 2a, but keep in mind that it now includes a TraversalSpec defining how to traverse the path.
    - VIM_PORT.retrieveProperties(PROP_COLLECTOR, propertyFilterSpecs);
      This line uses the retrieveProperties method to traverse the inventory to find the VMs and their properties, (as specified in the PropertySpec). This is the exact same call as in exercise 2a, the only change is that the propertyFilterSpec that is used as an argument now includes a traversal spec to find the objects.

      Note that like all methods related to managed objects, the method belongs to the service instance and the managed object it relates to is the first argument.

  After retrieving all the VMs, the function iterates through all the returned VMs and outputs their returned properties (name and powerstate were specified in the propertyspec, but the code here is more general if the propertyspec changes).

Step 4: Modify the Code

The code you'll be filling in for this lab is a part of the getVMTraversalSpec function. This code builds the TraversalSpec that defines how to locate a particular group of objects in the inventory (in this case all VMs in the VMfolder (which is all VMs). In a different function, this traversal spec is used as part of a propertyFilterSpec like the one you learned about in the previous exercise. In this case, the traversalSpec is part of the objectSpec defining which objects should be retrieved. The traversalSpec allows you to extend your propertyfilterspecs to locate properties of any object in the inventory.

In this case, the TraversalSpec you'll be building

The propertyFilterSpec you build should get the name and the powerstate of a virtualMachine.

In order to find properties of all virtual machine in the inventory, the TraversalSpec will need to start at a known object (in this case the root folder is specified as the starting object in the PropertyFilterSpec this traversal spec will be a part of). From there, the process of finding each additional object must be defined.

Based on the inventory tree above, the TraversalSpec needs to define how to get from the A folder (the root folder) to a Datacenter, and then how to get from a DataCenter to a folder (the VM folder). This is a 2 step traversal, so it will require 2 levels of TraversalSpecs.

The traversal will also need to account for nested folders. This can only occur between the datacenter and the VMs (since there are no additional folders allowed between the root folder and the datacenters).

The first TraversalSpec describes how to get from a datacenter to a folder

```
dataCenterToVMFolder

Name = "DataCenterToVMFolder" (What this traversalSpec is called when nested in another traversal spec)
Type = "Datacenter" (The type of object this traversal spec describes how to traverse to the next level of)
Path = "vmFolder" (the name of the property (found in the ref guide) to follow to get one level deeper)
Skip = false (whether to collect any properties from this object type)
SelectSet =
 sSpecs [ ]
  sSpec
  Name = "VisitFolders" (use the next spec when you arrive at a folder)
 (selectSet describes what traversalSpec to use next for each possible object type)
```

The second describes how to get from folders to other objects. These could be other folders or datacenters (because you don't need to traverse a VM to get it's properties, the end case of VirtualMachines is handled in the PropertyFilterSpec instead of in the TraversalSpec). To handle both cases, you need a 2 element SelectSet array. The second element will refer by name to the traversalSpec being defined to allow recursion into multiple levels of folders. The second element will be the traversal spec describing how to get from datacenters to folders.

```
traversalSpec

Name = "VisitFolders" (this is just what it is called in other specs, it doesn't match the object name)
Type = "Folder" (This time we want to find other objects from a folder)
Path = "childEntity" (whether you find VMs or DataCenters, they are under the childEntity prop)
Skip = false (whether to collect any properties from this object type)
setSelectSet(sSpecs);
SelectSet =
  sSpecArr [ ]
    sSpec
    Name = "VisitFolders" (recurse through folders)

    dataCenterToVMFolder (the object described above)

    Name = "DataCenterToVMFolder"
    Type = "Datacenter"
    Path = "vmFolder"
    Skip = false
    SelectSet =
      sSpecs [ ]
        sSpec
        Name = "VisitFolders" (use the next
        spec when you arrive at a folder)

  (selectSet describes what traversalSpec to use next for each possible object type)
```

4.1 Fill in the getVMTraversalSpec () function as follows:

```
public static TraversalSpec getVMTraversalSpec() {
   // Create a traversal spec that starts from the 'root' objects
   // and traverses the inventory tree to get to the VirtualMachines.
   // Build the traversal specs bottoms up
```

```
   // Traversal to get to the vmFolder from DataCenter
// Begin fill-in
// v v v v v v v
   TraversalSpec dataCenterToVMFolder = new TraversalSpec();
   dataCenterToVMFolder.setName("DataCenterToVMFolder");
   dataCenterToVMFolder.setType("Datacenter");
   dataCenterToVMFolder.setPath("vmFolder");
   dataCenterToVMFolder.setSkip(false);
   SelectionSpec sSpec = new SelectionSpec();
   sSpec.setName("VisitFolders");
   SelectionSpec[] sSpecs = new SelectionSpec[] { sSpec };
   dataCenterToVMFolder.setSelectSet(sSpecs);

   // TraversalSpec to get to the DataCenter from rootFolder
   TraversalSpec traversalSpec = new TraversalSpec();
   traversalSpec.setName("VisitFolders");
   traversalSpec.setType("Folder");
   traversalSpec.setPath("childEntity");
   traversalSpec.setSkip(false);
   SelectionSpec[] sSpecArr = new SelectionSpec[] { sSpec, dataCenterToVMFolder };
   traversalSpec.setSelectSet(sSpecArr);

   return traversalSpec;
// ^ ^ ^ ^ ^ ^ ^
// End fill-in
  }
```

Advanced variations (optional)

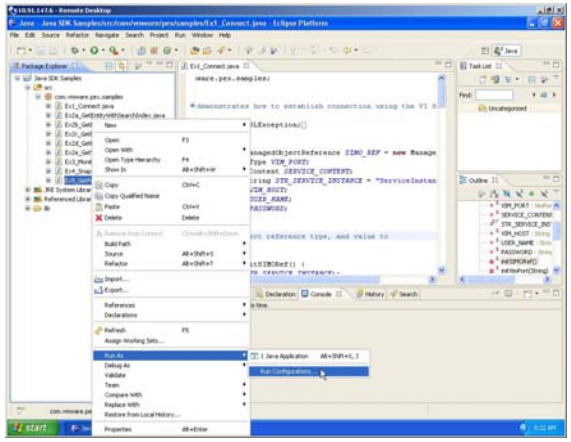
Examining the inventory tree, you'll see that there are many ways to reach VMs. Try to reach the VMs using a longer traversal spec that goes through the hostFolder and the ESX hosts on the way to the VMs. While this is less efficient, it provides an opportunity to understand that the inventory tree is actually more of a web. This approach may also be preferable in instances where only the VMs on specific hosts are being examined.

Step 5: Compiling and Running the Code

Once you've made the necessary changes to the code, compile and run the code. Despite your normal preferences when running and compiling code, please use the procedure in the lab to make it as easy as possible for the lab staff to diagnose any problems you have.

5.1 Verify that the program is complete and that you have filled in all the code sections specified in step 5.

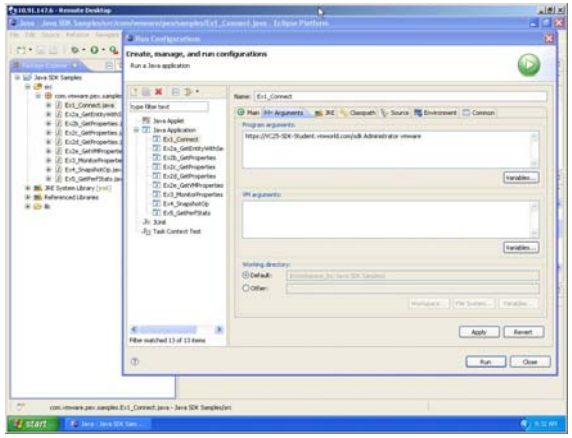
5.2 Right click on the file that you were working on in the Package Explorer, go to “Run As” and click “Run Configurations”. A new window will open up with a space for providing the arguments to the command. Recall that a number of the variables in the program are taken from the command line options.



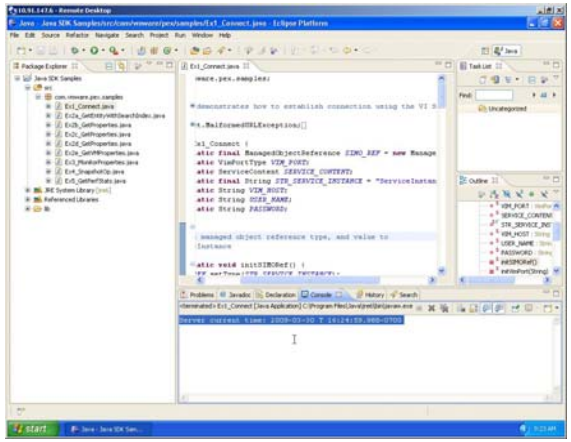
5.3 On the right (Run Configuration) window, confirm up top that it’s the correct file and click on the “Arguments” tab. This will also be the file that is highlighted in the pane on the left side of the Run Configuration window. If you do not have the correct file selected, you will re-run the last program.Input the arguments that that this program requires. You will need to input the following:

1) Virtual Center host (the VIM_HOST listed at your station)
2) Username (the USERNAME listed at your station)
3) Password (the PASSWORD listed at your station)
4) vminfo – *specifies which function in the code to run - in this case, it gets all VMs’ name and power states*

Usage: https://<VIM_HOST>/sdk <USERNAME> <PASSWORD> vminfo
Example : https://vc01-student01.vmworld.com/sdk sdk-student01 abc123 vminfo



5.4 Click Apply and Run Application at the bottom of the screen. You should see the correct output in the Console tab just below your code.



Optional - Hands-on Exercise 2c: Finding objects based on a property

Exercise 2a covered how to get the properties of objects using a PropertyFilterSpec if you already knew their MORefs (or could get them easily using the SearchIndex). Exercise 2b covered how you could find groups of objects in an inventory location using a TraversalSpec, allowing you to follow the methods in exercise 2a to get their properties.
This exercise describes how to find objects based on a property, rather than based on an inventory location.

Step 1: Understand how to find objects based on a property

There are only two basic ways to find objects in VI. The searchIndex can find objects based on datastore path, DNS name, inventory path, IP address, or UUID. A property collector using a Traversal Spec can find object based on inventory location. There are no other "built in" ways to find objects in VI. All other methods need to be derived from these.

Finding objects based on a property involves simply finding all the objects of a type using a traversal spec, and then only returning the ones that match a property.

While this may seem inefficient, keep in mind the search to do this needs only to include the property being selected for (minimizing the number of properties returned), and the VI SDK is optimized to allow these sorts of retrievals. Indexing the VI API to allow retrievals based on arbitrary properties would involve a huge number of indices that would need to be updated frequently.

This lab is nearly identical to 2b from a VI programming perspective. It merely illustrates an example of this concept, which is frequently a point of confusion for new VI programmers.

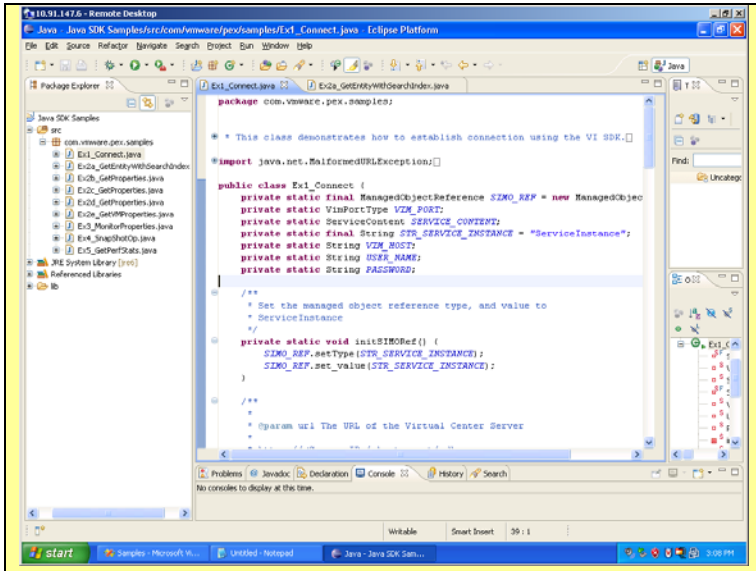
Step 2: Open the lab files for Ex2c_GetProperties.java

2.1 If you have not already opened eclipse, begin by opening “Java Lab” on the desktop.

Your IDE should be setup for you as shown below.

2.2 The source code is set for you on the left window in the Package Explorer under *src / com.vmware.pex.samples*. Double click the icon on the left-hand panel that corresponds to the code for this exercise number (the solutions for each exercises are also listed). Just selecting it may not work as it will only change your working file if you already have it open.

2.3 You should browse the code in the main window. Please do not compile or run the code until you get to that section of the instructions.



Step 3: Functional Walk-through for Ex2c_GetProperties.java

- initAll ()
  - initSIMORef ()
  - initVimPort ()
    - locator.setMaintainSession(true);
    - VIM_PORT = locator.getVimPort(new URL(url));
  - initServiceContent ()
    - SERVICE_CONTENT = VIM_PORT.retrieveServiceContent(SIMO_REF);
  - connect ()
    - VIM_PORT.login(SERVICE_CONTENT.getSessionManager(), uname, pword, null);
  - initPropertyCollector ()
    - PROP_COLLECTOR = SERVICE_CONTENT.getPropertyCollector();
  - initRootFolder
    - ROOT_FOLDER = SERVICE_CONTENT.getRootFolder();
- **getVMByName(vmName.trim())**

This is a function you will be writing or pasting for the lab. It is currently blank and the code is covered below.
getVMByName finds the MOref of the VM with the specified vmName and prints the MORef type and vmName. This requires using the PropertyCollector with a traversalSpec, rather than the SearchIndex used in exercise 2a, since there is no SearchIndex for finding things by VM name (only by DNS name).
The first part of the getVMByName function calls getVMTraversalSpec to create the traversal spec.

- getVMTraversalSpec()
  getVMTraversalSpec builds and returns a TraversalSpec that can be used later to find all VMs. Note that the TraversalSpec is just an object that describes how to traverse to find specific objects. It does not actually find the objects. Think of a Traversal spec like a map of how to find particular objects.

The next part creates the PropertyFilterSpec. It uses the same syntax as exercise 2a, except this time the ObjectSpec has a selectionSpec that includes the TraversalSpec used to find the VM objects. After building the PropertyFilterSpec, it is used in a retrieve properties call. This is the same call used in exercise 2a, but keep in mind that it now includes a TraversalSpec defining how to traverse the path.

- VIM_PORT.retrieveProperties(PROP_COLLECTOR, propertyFilterSpecs);
  This line uses the retrieveProperties method to traverse the inventory to find the VMs and their properties, (as specified in the PropertySpec). This is the exact same call as in exercise 2a, the only change is that the propertyFilterSpec that is used as an argument now includes a traversal spec to find the objects.

  Note that like all methods related to managed objects, the method belongs to the service instance and the managed object it relates to is the first argument.

After retrieving all the VMs, the function iterates through all the returned VMs and outputs the name of the VM that matches the name specified in the argument (which is taken indirectly from the command line).

- getVMName()
  getVMName finds and prints the name of a VM with the specified MORef. This is the same property collector code used in getVMInfo in exercise 2a, but this time the requested property is the name. This is used in the program just to verify that the MORef returned by GetVMByName matches the right VM.
- disconnect();

  The method disconnects the webservice and described in the sections above.

Step 4: Modify the Code

The code here simply involves a traversalSpec and PropertyFilterSpec nearly identical to the one in 2b. The code you'll be adding simply takes all the VM properties returned and finds the VM with a property matching the criteria to return.

4.1 Fill in the getVMByName () function as follows:

```
   public static ManagedObjectReference getVMByName(String vmName) {
      ManagedObjectReference retVal = null;
      try {
         TraversalSpec tSpec = getVMTraversalSpec();
         // Create Property Spec
         PropertySpec propertySpec = new PropertySpec();
         propertySpec.setAll(Boolean.FALSE);
         propertySpec.setPathSet(new String[] { "name" });
         propertySpec.setType("VirtualMachine");
         PropertySpec[] propertySpecs = new PropertySpec[] { propertySpec };

         // Now create Object Spec
         ObjectSpec objectSpec = new ObjectSpec();
         objectSpec.setObj(ROOT_FOLDER);
         objectSpec.setSkip(Boolean.TRUE);
         objectSpec.setSelectSet(new SelectionSpec[] { tSpec });
         ObjectSpec[] objectSpecs = new ObjectSpec[] { objectSpec };

         // Create PropertyFilterSpec using the PropertySpec and ObjectPec
         // created above.
         PropertyFilterSpec propertyFilterSpec = new PropertyFilterSpec();
         propertyFilterSpec.setPropSet(propertySpecs);
         propertyFilterSpec.setObjectSet(objectSpecs);

         PropertyFilterSpec[] propertyFilterSpecs = new PropertyFilterSpec[] { propertyFilterSpec };

         ObjectContent[] oCont = VIM_PORT.retrieveProperties(PROP_COLLECTOR,
               propertyFilterSpecs);
         if (oCont != null) {
           // System.out.println("ObjectContent Length : " + oCont.length);
           for (ObjectContent oc : oCont) {
               ManagedObjectReference mr = oc.getObj();
               // System.out.println("MOR Type : " + mr.getType());
               String vmnm = null;
               DynamicProperty[] dps = oc.getPropSet();
               if (dps != null) {
                  for (DynamicProperty dp : dps) {
                     // System.out.println(dp.getName() + " : " +
                     // dp.getVal());
                     vmnm = (String) dp.getVal();
                  }
```

```
               }

// Begin fill-in
// v v v v v v v
               // System.out.println("VM Name: " + vmnm);
               if (vmnm != null && vmnm.equals(vmName)) {
                  retVal = mr;
                  System.out.println("MOR Type : " + mr.getType());
                  System.out.println("VM Name: " + getVMName(mr));
                  break;
               }
            }
// ^ ^ ^ ^ ^ ^ ^
// End fill-in
         }
      } catch (Exception e) {
         e.printStackTrace();
      }

      return retVal;
   }
```

Variations (optional)

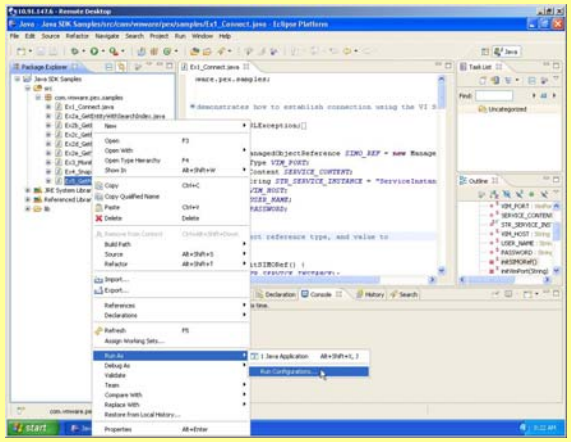
Try keying off alternative properties for this type of search. For instance, write a search that finds all VMs that have multiple CPUs (config.hardware.numCPUs), VMs that use a particular guest OS (guest.guestFamily), or VMs that are not using the right version of VMware tools (guest.toolsStatus).

Step 5: Compiling and Running the Code

Once you've made the necessary changes to the code, compile and run the code. Despite your normal preferences when running and compiling code, please use the procedure in the lab to make it as easy as possible for the lab staff to diagnose any problems you have.
5.1 Verify that the program is complete and that you have filled in all the code sections specified in step 5.

5.2 Right click on the file that you were working on in the Package Explorer, go to “Run As” and click “Run Configurations”. A new window will open up with a space for providing the arguments to the command. Recall that a number of the variables

in the program are taken from the command line options.

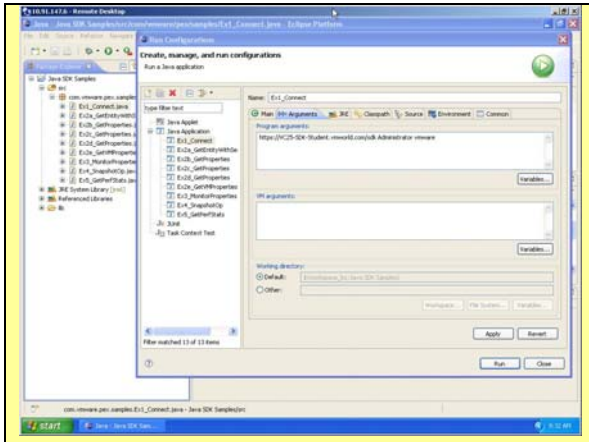


5.3 On the right (Run Configuration) window, confirm up top that it's the correct file and click on the "Arguments" tab. This will also be the file that is highlighted in the pane on the left side of the Run Configuration window. If you do not have the correct file selected, you will re-run the last program.

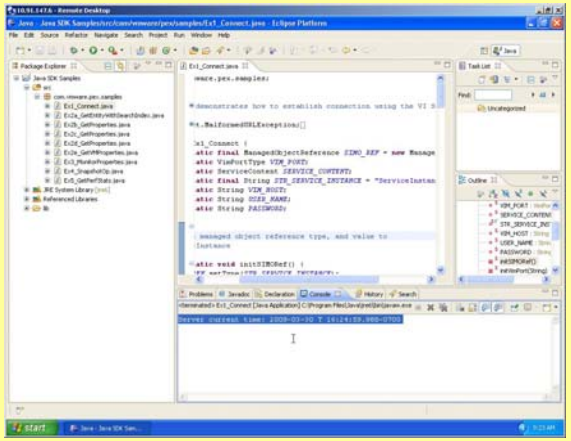
Input the arguments that that this program requires. You will need to input the following:

1) Virtual Center host (the VIM_HOST listed at your station)
2) Username (the USERNAME listed at your station)
3) Password (the PASSWORD listed at your station)
4) vminfo – *specifies which function in the code to run - in this case, it gets all VMs' managed object reference (moref)*
5) Entity name (the MY_VM_NAME listed at your station) – *name of the VM for which you're trying to get the moref*

Usage: https://<VIM_HOST>/sdk <USERNAME> <PASSWORD> vmmor <MY_VM_NAME>
Example : https://vc01-student01.vmworld.com/sdk sdk-student01 abc123 vmmor winsvc



5.4 Click Apply and Run Application at the bottom of the screen. You should see the correct output in the Console tab just below your code.



Hands-on Exercise 2d: Finding properties of multiple objects types at once

For efficiency reasons, it may sometimes be useful to return multiple types of objects in the same retrieveProperties () request. In addition, it may sometimes make sense to return multiple inventory locations of objects without returning all the objects of that type (for instance only VMs in the 3 critical SLA resource pools). This exercise covers how to aggregate these requests into one property filter request.

Step 1: Understand how to find properties of multiple object types at once

In order to handle multiple object types at once, several propertyFilterSpecs can be passed to retrieveProperties at once as an array. In this case, the code looks something like this:

Pseudocode	`PropertyFilterSpec[] pfSpecList = new PropertyFilterSpec[] {`
	`propertyFilterSpec1, vmPropertyFilterSpec2 };`
	ObjectContent[] oCont = *VIM_PORT*.retrieveProperties(*PROP_COLLECTOR*, pfSpecList);

At this point, it's a good time to point out the several ways you can retrieve properties of multiple objects at once:

Type of Request	**Method of aggregating request**
Multiple properties from a single object	multiple properties in PropertySpec PathSet [ ] property array PropertyFilterSpec using no traversal spec
Multiple properties from Objects of the same type in similar inventory locations (this could be different locations with the same traversal spec for nested folders or resource groups)	multiple properties in PropertySpec PathSet [ ] property array PropertyFilterSpec with a traversal spec
Multiple objects (potentially different types) in the same inventory traversal (same inventory starting point, but potentially different paths)	multiple propertySpecs in PropertyFilterSpec PropSet [ ] property array.
Multiple objects (potentially different types) at different inventory locations traversable from the same starting point	multiple SelectionSpecs and/or TraversalSpecs in the ObjectSpec selectSet [ ] property
Multiple objects (potentially different types) reachable from	multiple objectSpecs in the PropertyFilterSpec objectSet [ ]

different starting locations for traversals	property array
Multiple objects (potentially different types) reachable from different starting locations for traversals, returned as a number of different groups of results (one for each propertyfilterspec) from the same SOAP request	multiple propertyfilterSpecs as retrieveProperties () specset [ ] arguments.

An exampled of the performance implications of making individual requests as opposed to aggregating requests is covered in exercise 2e.

In this lab, you'll make a request of the last type. Because you're starting from the root folder, this also would also fit the case of multiple objects of different types traversable from the same starting point. While it would be possible to make the specification that way with multiple traversal specs in the objectSpec's selectSet[] property, doing it that way would result in objects of different types returned in a random order. This would require seperating them into the different object types for most types of later processing. Using multiple propertyfilterSpecs has the advantage of returning a separate objectSpec for each propertFilterSpec, separating the different object types neatly.

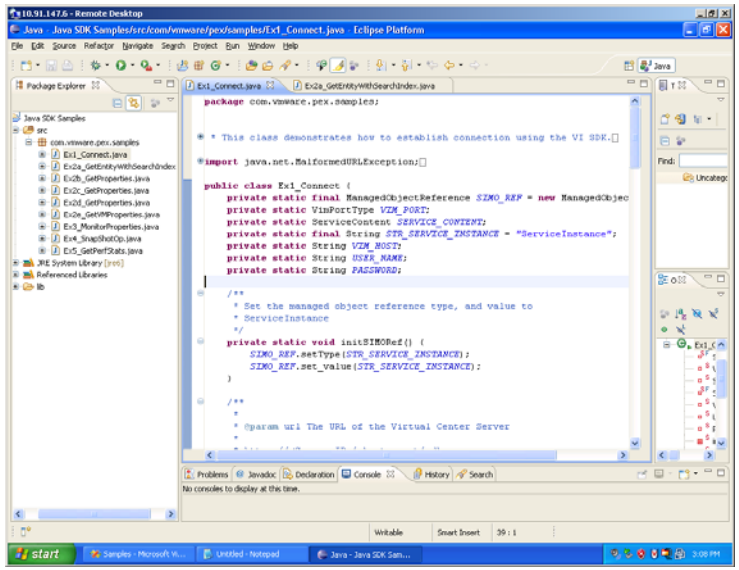
Step 2: Open the lab files for Ex2d_GetProperties.java

2.1 If you have not already opened eclipse, begin by opening “Java Lab” on the desktop.

Your IDE should be setup for you as shown below.

2.2 The source code is set for you on the left window in the Package Explorer under *src / com.vmware.pex.samples*. Double click the icon on the left-hand panel that corresponds to the code for this exercise number (the solutions for each exercises are also listed). Just selecting it may not work as it will only change your working file if you already have it open.

2.3 You should browse the code in the main window. Please do not compile or run the code until you get to that section of the instructions.



Step 3: Functional Walk-through for GetProperties.java

- initAll ()
  - initSIMORef ()
  - initVimPort ()
    - locator.setMaintainSession(true);
    - VIM_PORT = locator.getVimPort(new URL(url));
  - initServiceContent ()
    - SERVICE_CONTENT = VIM_PORT.retrieveServiceContent(SIMO_REF);
  - connect ()
    - VIM_PORT.login(SERVICE_CONTENT.getSessionManager(), uname, pword, null);
  - initPropertyCollector ()
    - PROP_COLLECTOR = SERVICE_CONTENT.getPropertyCollector();
  - initRootFolder
    - ROOT_FOLDER = SERVICE_CONTENT.getRootFolder();

- **getHostAndVMNames();**
  This is a function you will be writing or pasting for the lab. It is currently blank and the code is covered below.
  getHostAndVMNames follows the same pattern as the exercises in 2b, but with the additional ability to make a single webservices call that returns properties of multiple types of objects. The structure of this function is similar to the pattern in 2b: build a TraversalSpec, PropertySpec, and ObjectSpec and combine them into a PropertyFilterSpec, then call retrieveProperties() on that spec. The difference here is that two different types of objects are being called (though you'd use the same pattern for the same object type in two different inventory locations). Because of this, the process of building the TraversalSpec, PropertySpec, and ObjectSpec and combine them into a PropertyFilterSpec will need to be repeated twice-- once for each object type. The code here reuses the code for doing this for a VM and adds some very similar code for doing the same thing for an ESX host.
  - getHostSystemTraversalSpec();
    getHostSystemTraversalSpec builds and returns a TraversalSpec that can be used later to find all ESX hosts. This is very similar to the TraversalSpec used to find VMs in exercise 2b, but follows a slightly different path through the inventory that leads to the hosts.

  At the end of the process, both of the propertyFilterSpecs are combined in an array which is used as an argument to a single retrieveProperties call.

Step 4: Modify the code

The code here is essentially just a combination of the host and VM propertyFilter specs in exercise 2b. You simply need to combine them in getHostAndVMNames and pass both as an array to the retrieveProperties method.

4.1 Fill-in the getHostAndVMNames() function as follows:

```
    public static void getHostAndVMNames() {
    try {
// Begin fill-in
// v v v v v v v
       // Create Property Filter Spec for HostSystem
       TraversalSpec tSpec = getHostSystemTraversalSpec();
       // Create Property Spec
       PropertySpec propertySpec = new PropertySpec();
       propertySpec.setAll(Boolean.FALSE);
       propertySpec.setPathSet(new String[] { "name" });
       propertySpec.setType("HostSystem");
       PropertySpec[] propertySpecs = new PropertySpec[] { propertySpec };

       // Now create Object Spec
       ObjectSpec objectSpec = new ObjectSpec();
```

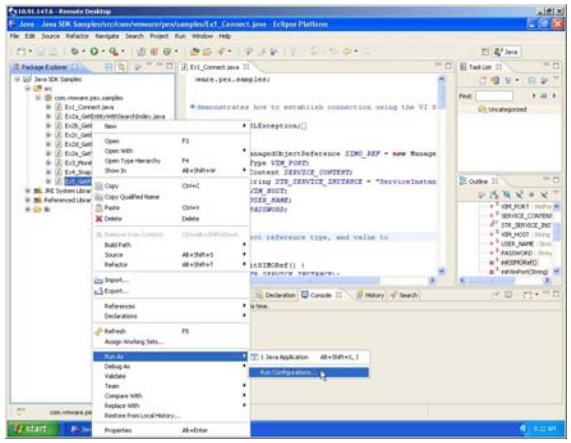
```
        objectSpec.setObj(ROOT_FOLDER);
        objectSpec.setSkip(Boolean.TRUE);
        objectSpec.setSelectSet(new SelectionSpec[] { tSpec });
        ObjectSpec[] objectSpecs = new ObjectSpec[] { objectSpec };

        // Create PropertyFilterSpec using the PropertySpec and ObjectPec
        // created above.
        PropertyFilterSpec propertyFilterSpec = new PropertyFilterSpec();
        propertyFilterSpec.setPropSet(propertySpecs);
        propertyFilterSpec.setObjectSet(objectSpecs);

        // Create Property Filter Spec for VirtualMachine
        TraversalSpec vmTraversalSpec = getVMTraversalSpec();
        // Create Property Spec
        PropertySpec vmPropertySpec = new PropertySpec();
        vmPropertySpec.setAll(Boolean.FALSE);
        vmPropertySpec.setPathSet(new String[] { "name" });
        vmPropertySpec.setType("VirtualMachine");
        PropertySpec[] vmPropertySpecs = new PropertySpec[] { vmPropertySpec };

        // Now create Object Spec
        ObjectSpec vmObjectSpec = new ObjectSpec();
        vmObjectSpec.setObj(ROOT_FOLDER);
        vmObjectSpec.setSkip(Boolean.TRUE);
        vmObjectSpec.setSelectSet(new SelectionSpec[] { vmTraversalSpec });
        ObjectSpec[] vmObjectSpecs = new ObjectSpec[] { vmObjectSpec };

        // Create PropertyFilterSpec using the PropertySpec and ObjectPec
        // created above.
        PropertyFilterSpec vmPropertyFilterSpec = new PropertyFilterSpec();
        vmPropertyFilterSpec.setPropSet(vmPropertySpecs);
        vmPropertyFilterSpec.setObjectSet(vmObjectSpecs);

        // Add HostSystem and VirtualMachine property filter specs to
        // property filter spec list
        PropertyFilterSpec[] pfSpecList = new PropertyFilterSpec[] {
             propertyFilterSpec, vmPropertyFilterSpec };

// ^ ^ ^ ^ ^ ^ ^
// End fill-in
```

```
        ObjectContent[] oCont = VIM_PORT.retrieveProperties(PROP_COLLECTOR,
              pfSpecList);
        if (oCont != null) {
          // System.out.println("ObjectContent Length : " + oCont.length);
          for (ObjectContent oc : oCont) {
             DynamicProperty[] dps = oc.getPropSet();
             ManagedObjectReference mor = oc.getObj();
             if (dps != null) {
                for (DynamicProperty dp : dps) {
                  System.out.println(mor.getType() + " "
                        + dp.getName() + " : " + dp.getVal());
                }
             }
          }
        }
     } catch (Exception e) {
        e.printStackTrace();
     }
  }
```

Step 5: Compiling and Running the Code

Once you've made the necessary changes to the code, compile and run the code. Despite your normal preferences when running and compiling code, please use the procedure in the lab to make it as easy as possible for the lab staff to diagnose any problems you have.
5.1 Verify that the program is complete and that you have filled in all the code sections specified in step 5.

5.2 Right click on the file that you were working on in the Package Explorer, go to “Run As” and click “Run Configurations”. A new window will open up with a space for providing the arguments to the command. Recall that a number of the variables in the program are taken from the command line options.

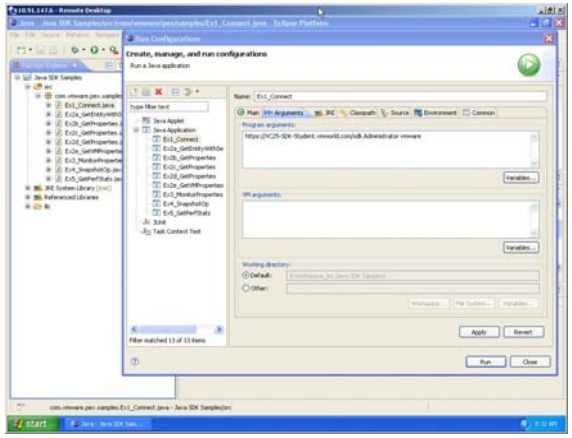


5.3 On the right (Run Configuration) window, confirm up top that it's the correct file and click on the "Arguments" tab. This will also be the file that is highlighted in the pane on the left side of the Run Configuration window. If you do not have the correct file selected, you will re-run the last program.

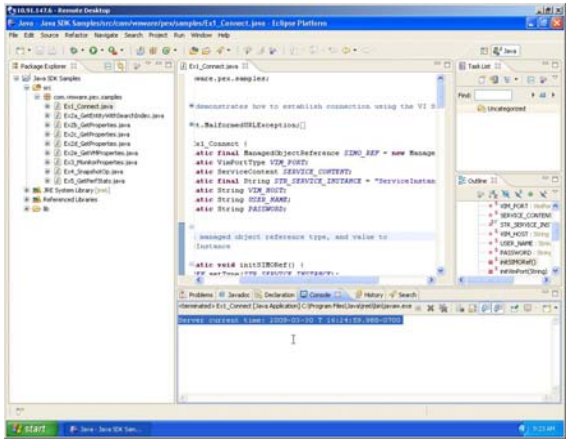
Input the arguments that that this program requires. You will need to input the following:

1) Virtual Center host (the VIM_HOST listed at your station)
2) Username (the USERNAME listed at your station)
3) Password (the PASSWORD listed at your station)
4) hostvminfo - *To retrieve the names of the VMs and hosts*

Usage: https://<VIM_HOST>/sdk <USERNAME> <PASSWORD> hostvminfo
Example : https://vc01-student01.vmworld.com/sdk sdk-student01 abc123 hostvminfo



5.4 Click Apply and Run Application at the bottom of the screen. You should see the correct output in the Console tab just below your code.



Optional - Hands-on Exercise 2e: Increasing Performance of PropertyCollector Calls

A key issue in VI SDK programming is using the Property Collector correctly to maximize performance. Efficient use of the property collector not only minimizes the performance impact of calls on VC, it also speeds up the execution of your programs. In many cases development environments are fairly small. However VMware customer environments can have thousands of VMs. Inefficient use of the property collector can result in queries that run in less than a second in a development environment taking minutes or longer at a production customer site. Aggregating property collector calls together has resulted in query-time improvements of 100x or more for some developers.

Step 1: Understand the performance of PropertyCollector Calls

This lab only covers one possible option for aggregating multiple calls together. In most cases, other types of call aggregation may be more appropriate. The table in exercise 2d covers several examples of call aggregation for the Property Collector. In most cases, using the first listed method on the table that applies to a given case is going to result in the most efficient processing. For instance, while you could do a traversal and property collection for 10 different properties of the same object in 10 different propertyCollectorSpecs, it would generally be much better to use one PropertyCollectorSpec with 10 properties listed in the PropertySpec. Keep in mind that the efficiency isn't just derived from the number of SOAP requests; it is also derived from back-end performance optimization in VC that may be able to optimize certain types of data queries.

It is also important to point out that too much aggregation can be a bad thing. The benefits of using aggregation for large numbers of properties or objects being collected in tight loops are almost always a positive thing. These sorts of tight loops result in many more SOAP calls than necessary and generally can always be done better by grouping the properties, objects, or traversals in an array. However, in some cases slightly less efficient methods are better because they result in less post-processing or a better program architecture.

Efficiency in PropertyCollectorCalls is also related to the amount of data being sent back. It is also important to consider limiting the data, especially for large environments. For instance, only properties that will be used should be requested. Similarly, if there is a way to limit the collector to a sub-inventory of objects that are interesting, that can be a benefit as well.

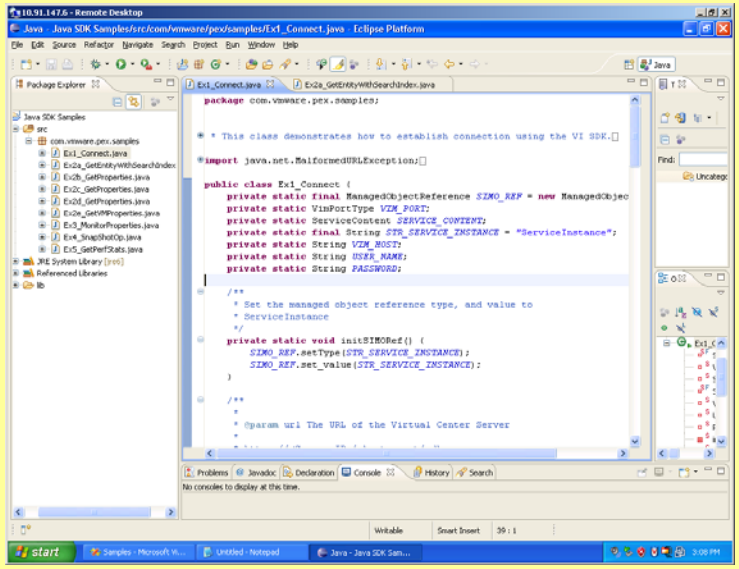
Step 2: Open the lab files for Ex2e_GetProperties.java

2.1 If you have not already opened eclipse, begin by opening “Java Lab” on the desktop.

Your IDE should be setup for you as shown below.

2.2 The source code is set for you on the left window in the Package Explorer under *src / com.vmware.pex.samples*. Double click the icon on the left-hand panel that corresponds to the code for this exercise number (the solutions for each exercises are also listed). Just selecting it may not work as it will only change your working file if you already have it open.

2.3 You should browse the code in the main window. Please do not compile or run the code until you get to that section of the instructions.



Step 3: Functional Walk-through for GetProperties.java

Because this exercise simply compares two methods of doing something, you will be writing main().

- initAll ()
  - initSIMORef ()
  - initVimPort ()
    - locator.setMaintainSession(true);
    - VIM_PORT = locator.getVimPort(new URL(url));
  - initServiceContent ()
    - SERVICE_CONTENT = VIM_PORT.retrieveServiceContent(SIMO_REF);
  - connect ()

        - VIM_PORT.login(SERVICE_CONTENT.getSessionManager(), uname, pword, null);
    - initPropertyCollector ()
        - PROP_COLLECTOR = SERVICE_CONTENT.getPropertyCollector();
    - initRootFolder
        - ROOT_FOLDER = SERVICE_CONTENT.getRootFolder();
- st = System.currentTimeMillis();
- getVMInfo();
  This is the same code as getVMInfo from exercise 2b.
- et = System.currentTimeMillis();
- st = System.currentTimeMillis();
- initVMMorList();
  initVMMorList gets a list of the MORefs for all VMs. This is very similar to the code for GetVMInfo in exercise 2b.
- getVMProperty(vmMor, powerState);
  getVMProperty takes a managed object and a string representing the name of the property to get. This very similar to the code for GetVMInfo in exercise 2a, but is generic instead of specifying a specific property.
  This function is used in the code to get the power state for VMs, one retreiveProperties call at a time.
- et = System.currentTimeMillis();

Step 4: Modify Code

In this exercise, the code you'll modify is in main(). The code here doesn't involve any new VI SDK programming concepts. In this case, the goal is simply to show the difference between calling a property that aggragates a number of properties and one that asks for each property seperately. Even in a small environment, the difference between these two methods can be as much as 10x.

4.1 Fill in the main() function as follows:

```
    public static void main(String[] args) {
    // This is to accept all SSL certifcates by default.
    System.setProperty(
          "org.apache.axis.components.net.SecureSocketFactory",
          "org.apache.axis.components.net.SunFakeTrustSocketFactory");
    if (args.length < 3) {
      printUsage();
    } else {
      try {
// Begin fill-in
```

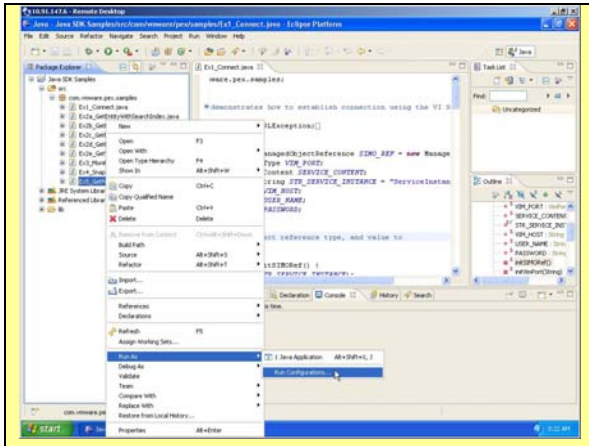
```
// v v v v v v v

          VIM_HOST = args[0];
          USER_NAME = args[1];
          PASSWORD = args[2];

          initAll();

          System.out
                .println("*****************************************************************");
          long st = System.currentTimeMillis();
          getVMInfo();
          long et = System.currentTimeMillis();
          System.out
                .println("\nTotal time (msec) to retrieve the properties of all VMs in one call: "
                      + (et - st));
          System.out
                .println("\n*****************************************************************");

          System.out
                .println("\n*****************************************************************");
          st = System.currentTimeMillis();
          initVMMorList();
          Iterator<ManagedObjectReference> iter = VM_MOR_LIST.iterator();
          StringBuilder sb = new StringBuilder();
          String name = "name";
          String powerState = "runtime.powerState";

          while (iter.hasNext()) {
            ManagedObjectReference vmMor = iter.next();
            String vmName = (String) getVMProperty(vmMor, name);
            sb.append(vmName);
            VirtualMachinePowerState vmPs = (VirtualMachinePowerState) getVMProperty(
                  vmMor, powerState);
            sb.append(" : ");
            sb.append(vmPs);
            sb.append("\n");
          }
          et = System.currentTimeMillis();
          System.out.println(sb.toString());
          System.out
```

```
                .println("\nTotal time (msec) to retrieve the properties of all VMs individually: "
                      + (et - st));
            System.out
                .println("\n*************************************************************");

      }

// ^ ^ ^ ^ ^ ^ ^
// End fill-in
} catch (Exception e) {
          e.printStackTrace();
      } finally {
          try {
            disconnect();
          } catch (Exception e) {
            e.printStackTrace();
          }
      }
   }
  }
```

Step 5: Compiling and Running the Code

Once you've made the necessary changes to the code, compile and run the code. Despite your normal preferences when running and compiling code, please use the procedure in the lab to make it as easy as possible for the lab staff to diagnose any problems you have.
5.1 Verify that the program is complete and that you have filled in all the code sections specified in step 5.

5.2 Right click on the file that you were working on in the Package Explorer, go to “Run As” and click “Run Configurations”. A new window will open up with a space for providing the arguments to the command. Recall that a number of the variables in the program are taken from the command line options.

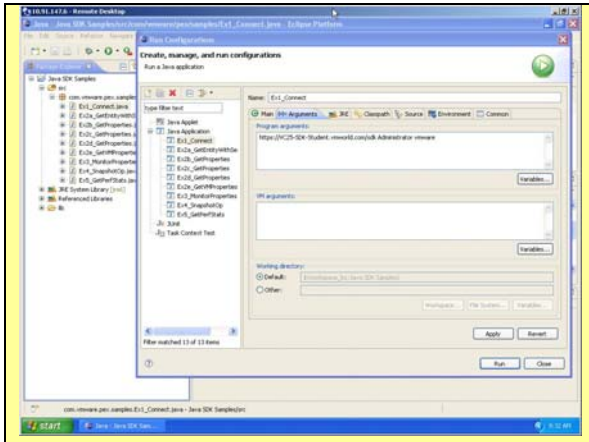


5.3 On the right (Run Configuration) window, confirm up top that it's the correct file and click on the "Arguments" tab. This will also be the file that is highlighted in the pane on the left side of the Run Configuration window. If you do not have the correct file selected, you will re-run the last program.

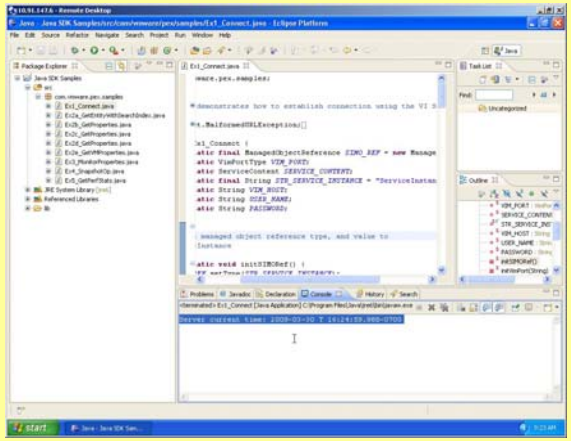
Input the arguments that that this program requires. You will need to input the following:

1) Virtual Center host (the VIM_HOST listed at your station)
2) Username (the USERNAME listed at your station)
3) Password (the PASSWORD listed at your station)

Usage: https://<VIM_HOST>/sdk <USERNAME> <PASSWORD>
Example : https://vc01-student01.vmworld.com/sdk sdk-student01 abc123



5.4 Click Apply and Run Application at the bottom of the screen. You should see the correct output in the Console tab just below your code.



Hands-on Exercise 3: Tracking Property Changes

Exercise 2 covered how to use the Property Collector to find the properties of managed objects. At this point you should have a clear understanding of what they property collector is and how to use it.
This exercise introduces another use for the property collector: tracking changes in objects and properties. Tracking changes is useful in a number of situations. Since properties typically change only rarely (recall that performance stats are handled elsewhere), tracking changes to objects is usually easier and more efficient than collecting the properties again. Also, tracking changes is an easy way to track the status of tasks that have been triggered.

Step 1: Understand how to track property changes

Property changes are tracked using a managed object called a PropertyFilter. As you may have noticed when examining the PropertyCollector, it contains an array of PropertyFilters. These PropertyFilters are all the filters a given client session has set up to note property changes. A property filter is created by calling the createFilter method (associated with the PropertyCollector). createFilter uses a PropertyFilterSpec to specify what properties to filter. This is exactly the same object used throughout exercise 2 to specify what properties to retrieve the value of. In fact, a call to retrieveProperties is essentially the same as calling createFilter, collecting the current state, and then deleting the filter.
Once a filter is created, the updates can be checked in two ways. The CheckForUpdates () method is a poll-based method that gets the latest updates at the time it is called. The WaitForUpdates () method is a blocking method that requests the latest updates and doesn't return until one of the filtered properties changes.
The PropertyFilterSpec selects properties exactly as in exercise 2.
The CreateFilter() method has two arguments: the PropertyFilterSpec specifying the properties to filter, and a boolean specifying whether to return the entire object property that changed or just the nested part of it that changed.
The CheckForUpdates() and WiatForUpdates() methods each have 2 arguments: the first is the PropertyFilter they are monitoring for changes, and the second is the version of the changes the client currently knows (so only more recent changes can be passed back).
WaitForUpdates and CheckForUpdates return an UpdateSet, which is an array of the PropertyFilterUpdates corresponding to each filter it is monitoring for.

1.1 Look up the PropertyFilterUpdate object in the Reference Guide.

The PropertyFilterUpdates will indicate the results of a particular filter, returning an ObjectUpdate representing the object changes to the filtered object. The *changeSet* property of the ObjectUpdate contains information about the actual Property changed (a PropertyChange object). Examining the value property of the PropertyChange object provides the new value of the changed field. In the exercise, all this is accomplished by a series of foreach loops that go through all the various

changed filters, objects, and properties until they find the correct one. Each of these object provides much more data on the type of changes, which can be researched in the SDK reference guide.

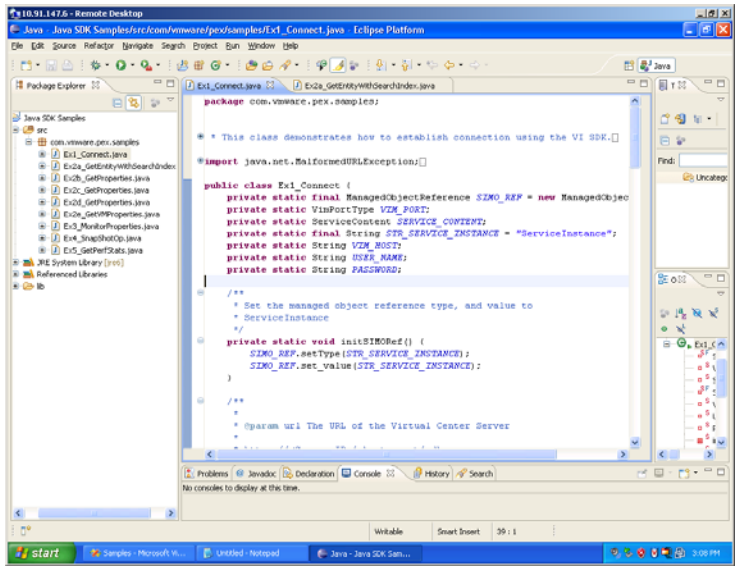
Step 2: Open the lab files for Ex3_MonitorProperties.java

2.1 If you have not already opened eclipse, begin by opening “Java Lab” on the desktop.

Your IDE should be setup for you as shown below.

2.2 The source code is set for you on the left window in the Package Explorer under *src / com.vmware.pex.samples*. Double click the icon on the left-hand panel that corresponds to the code for this exercise number (the solutions for each exercises are also listed). Just selecting it may not work as it will only change your working file if you already have it open.

2.3 You should browse the code in the main window. Please do not compile or run the code until you get to that section of the instructions.



Step 3: Functional Walk-through for MonitorProperties.java

- initAll ()

- initSIMORef ()
- initVimPort ()
  - locator.setMaintainSession(true);
  - VIM_PORT = locator.getVimPort(new URL(url));
    ((org.apache.axis.client.Stub)VIM_PORT).setTimeout(1200000);
    There is one line here that is different from previous versions. Note that here the web service port timeout for the axis client is set to a very large value. The reason for this is that we will use the waitForUpdates call. The waitForUpdates call makes an HTTP request and will not receive an answer from the server until a monitored property changes. Because this could be a long time, the timeout needs to be extended to make sure that the http connection does not timeout while waitForUpdates is still waiting.
- initServiceContent ()
  - SERVICE_CONTENT = VIM_PORT.retrieveServiceContent(SIMO_REF);
- connect ()
  - VIM_PORT.login(SERVICE_CONTENT.getSessionManager(), uname, pword, null);
- initPropertyCollector ()
  - PROP_COLLECTOR = SERVICE_CONTENT.getPropertyCollector();
- initRootFolder
  - ROOT_FOLDER = SERVICE_CONTENT.getRootFolder();

- **monitorProperties(propArr);**
  This is a function you will be writing or pasting for the lab. It is currently blank and the code is covered below.
  - createPropertyFilter(propArr);
    createPropertyFilter takes the name of a property to monitor (this is indirectly from the command line) and creates a propertyFilter that will flag changes to that property in any VM on the server.
    The code to do this is nearly the same as the code to get the property in the first place. Compare this with GetVMInfo () in GetProperties.java. The only difference is that the VI SDK method used is createFilter instead of retrieveProperties. Essentially, the code here creates a PropertyFilterSpec that traverses the inventory to cover all VMs and selects the specified property.
    - getVMTraversalSpec()
      This is the same function used in a number of the previous exercises to traverse the VMs.
    - VIM_PORT.createFilter(PROP_COLLECTOR, propertyFilterSpec, false);
      The propertyFilterSpec used here is just like the ones in the previous exercises. The only thing new here is the use of createFilter instead of retrieveProperties. As with all calls related to managed objects, the method belongs to the service instance (since the managed object isn't local), and it includes a reference

to the managed object. The "false" value for the last argument indicates that partial changes should return the whole property, not just the partial change.

- VIM_PORT.waitForUpdates(PROP_COLLECTOR, version);
  This line uses the waitForUpdates method to wait until the propertyCollector has updated past the specified version. The looping structure around this is a lot like the main event loop in a typical interactive program. The waitForUpdates method will wait for a change in the property collector. This is handled in a bit of an unusual manner, since this is a web service call. In this case, the waitForUpdates method makes a SOAP call to the VI API. However, no SOAP response is sent until an update occurs. This essentially allows polling over HTTP. In order to do this effectively, the HTTP timeout needs to be extended. This is handled in a modification to initVimPort above.

  In a more complex program, this sort of polling would be handled in a listener thread, allowing the rest of the execution to continue.

  Note that like all methods related to managed objects, the method belongs to the service instance and the managed object it relates to is the first argument.

- updateSet.getVersion();
  When an update occurs, the program checks the version of the update. This is simply an accessor function to the version property of the updateSet object. In this case, the version is simple printed. In a more complex program, it may be important to make sure that the data received is the latest version, and not simply a time delayed older version.
- updateSet.getFilterSet();
  This line gets the filterSet data from the updateSet object, this is just an accessor function for updateSet. Essentially the code in this and the next few steps is unwrapping the changes that got identified by the filter. As shown in the reference guide, the updateSet object returned by CheckForUpdates and WaitForUpdates contains an array called filterSet that is of type PropertyFilterUpdate. Objects in that array contain an array called objectSet of type ObjectUpdate. Those objects actually
- for (PropertyFilterUpdate pfu : pfuArr) {
  This line begins a loop iterating through the propertyFilterUpdates. Each propertyFilterUpdate represents a specific set of managed object updates
  - ObjectUpdate[] ouArr = pfu.getObjectSet();
    The ObjectSet of each propertyfilter represents the array of each particular object that had updates for that set.
  - for (ObjectUpdate ou : ouArr) {
    This line iterates through the objectSet, going through one changed object for each loop.

- ManagedObjectReference mor = ou.getObj();
  The MORef here is the MORef of each changed object, allowing the program to identify each object..
- PropertyChange[] pcArr = ou.getChangeSet();
  For each object, any number of property changes could have occurred. This gets an array indicating the set of property changes. The array is later iterated through to list each changed property.

o pfu.getFilter()destroyPropertyFilter(vmFilter);
This line simply destroys the property filter now that its work is done.

Step 4: Modify Code

Because the propertyFilterSpec object is the same as before, the code sample here concentrates on how to gather the updates. The code here creates a filter, and then calls wiatForUpdates in a loop. Every time an update occurs, it is printed, and the loop iterates so that the program waits for a new update. Note that waitForUpdates is ideal for this use because is avoids unnecessary polling (which might occur with checkForUpdates).

4.1 Fill-in the code in monitorProperties as shown below:

```
    public static void monitorProperties(String[] propArr) {
    ManagedObjectReference vmFilter = null;
    try {
// Begin fill-in
// v v v v v v v
        vmFilter = createPropertyFilter(propArr);
        System.out.println("vmFilter : " + vmFilter.getType() + " : "
              + vmFilter.get_value());
        // At the time of first invocation, we don't know the initial
        // version.
        // So, we start with empty string. When the waitForUpdates returns,
        // the UpdateSet contains the version, which needs to be used in the
        // invocations to track changes from that point.
        String version = "";
        System.out.println("--------------------");
        while (true) {
           UpdateSet updateSet = VIM_PORT.waitForUpdates(PROP_COLLECTOR, version);
           version = updateSet.getVersion();
           System.out.println("Update received: " + version);
```

```
            PropertyFilterUpdate[] pfuArr = updateSet.getFilterSet();
            for (PropertyFilterUpdate pfu : pfuArr) {
                System.out.println("-------------------");
                System.out.println("Filter val: "
                        + pfu.getFilter().getType() + " : "
                        + pfu.getFilter().get_value());
                ObjectUpdate[] ouArr = pfu.getObjectSet();
                for (ObjectUpdate ou : ouArr) {
                    // System.out.println("Change type: " + ou.getKind());
                    ManagedObjectReference mor = ou.getObj();
                    System.out.println(mor.getType() + " : "
                            + getManagedEntityName(mor));
                    System.out.println();
                    PropertyChange[] pcArr = ou.getChangeSet();
                    for (PropertyChange pc : pcArr) {
                        System.out.println("     Property Name: "
                              + pc.getName());
                        // System.out.println("Op: " + pc.getOp());
                        System.out.println("     Property Val: " + pc.getVal()
                              + "\n");
                    }
                }
                System.out.println("-------------------");
            }
        }
// ^ ^ ^ ^ ^ ^ ^
// End fill-in
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (vmFilter != null) {
            destroyPropertyFilter(vmFilter);
        }
    }
 }
```

Variations (optional)

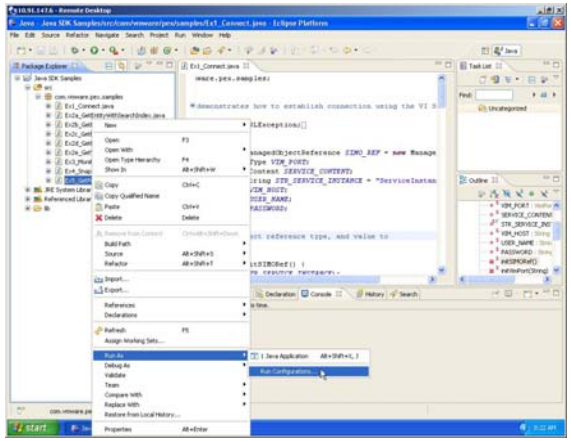
Modify the program to just poll for the value a specific amount of time after it runs. You can do this by removing the loop

and using checkForUpdates with the same syntax after a sleep statement.

Step 5: Compiling and Running the Code

Once you've made the necessary changes to the code, compile and run the code. Despite your normal preferences when running and compiling code, please use the procedure in the lab to make it as easy as possible for the lab staff to diagnose any problems you have.
5.1 Verify that the program is complete and that you have filled in all the code sections specified in step 5.

5.2 Right click on the file that you were working on in the Package Explorer, go to “Run As” and click “Run Configurations”. A new window will open up with a space for providing the arguments to the command. Recall that a number of the variables in the program are taken from the command line options.



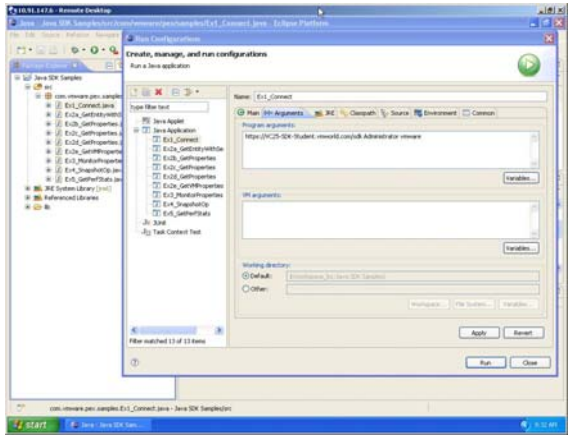
5.3 On the right (Run Configuration) window, confirm up top that it’s the correct file and click on the “Arguments” tab. This will also be the file that is highlighted in the pane on the left side of the Run Configuration window. If you do not have the correct file selected, you will re-run the last program.

Input the arguments that that this program requires. You will need to input the following:

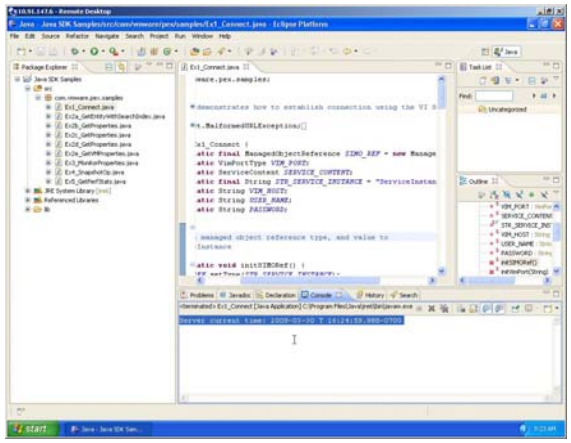
1) Virtual Center host (the VIM_HOST listed at your station)
2) Username (the USERNAME listed at your station)

3) Password (the PASSWORD listed at your station)
4) name,runtime.powerState,summary.runtime.maxCpuUsage – *VM properties that you want to get -Comma separated values with no white space after the commas.*

Usage: https://<VIM_HOST>/sdk <USERNAME> <PASSWORD> name,runtime.powerState,summary.runtime.maxCpuUsage
Example : https://vc01-student01.vmworld.com/sdk sdk-student01 abc123 name,runtime.powerState,summary.runtime.maxCpuUsage



5.4 Click Apply and Run Application at the bottom of the screen. You should see the correct output in the Console tab just below your code.



5.5 The application will output the current values being monitored, then waits in an endless loop for the state changes you specified. In order to see any output, you'll need to make state changes to the objects that match they properties you are monitoring. For instance, turn the VM on and off, since you are monitoring the power state.

Make sure the VM is on when you are done, since you will need it on for the next exercise.

Please do **NOT** change the name of the VM. It will make it harder to reset the lab setup for the next group of students.

Hands-on Exercise 4: Tasks and Changes to vSphere

Exercise 4 teaches how to handle snapshots. This is an introduction to how to make changes to managed objects in a vSphere environment. Many different types of changes are possible in a vSphere environment, and this exercise is only an introduction to the process.

Because most types of changes in the vSphere environment take time, changes frequently result in the creation of a task. Tasks are managed objects that can be used to track the progress and results of a requested change.

Step 1: Understanding Reconfiguration, Snapshots, and Tasks

Understanding vSphere Changes

There are a number of types of VI Reconfiguration operations. These typically fall into a few basic categories:

Acquire tickets for other APIs and services

Add, create, or clone objects

User and permission control

Cancel operations

Check capabilities and compatibility

Perform operations in a VM or host (customize a VM, modify VMFS, or manage host configuration)

Delete or destroy objects

Disable or enable features and capabilities

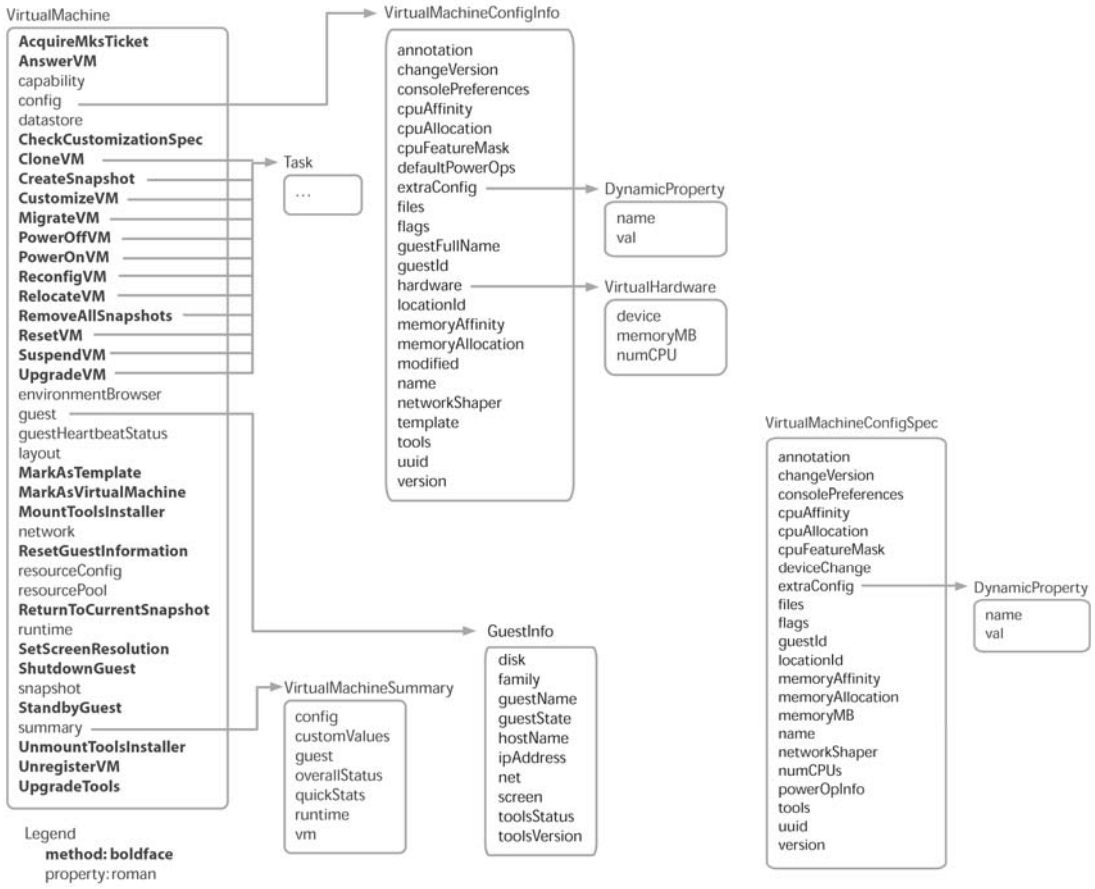
Move objects

Reconfigure objects

Control powerstate

Changes to vSphere are always made using a method associated with a particular Managed Object. In most cases, the arguments to these requests are fairly simple. However, for some types of changes, like object reconfigurations, the arguments must define the exact properties to be changed and may require objects nearly as complex as the managed object being changed.

For example, below are Diagrams for **VirtualMachine** and **VirtualMachineConfigSpec**. Note that **VirtualMachineConfigSpec** has fields corresponding to each field of the **VirtualMachine** *config* property:



1.1 Look up the VirtualMachine Object in the Reference Guide and explore some of the available methods.

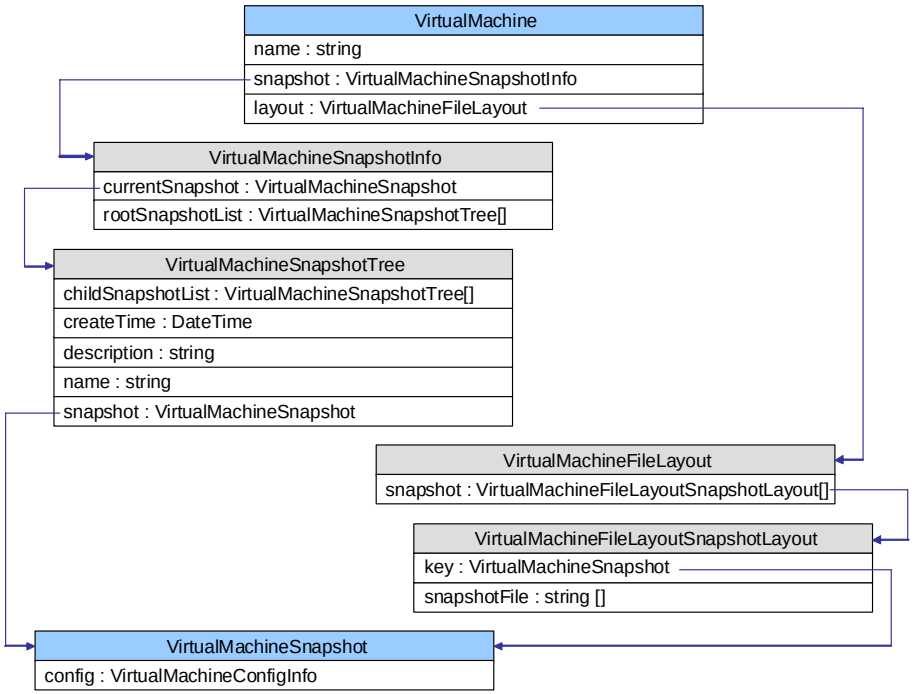
In this lab, we'll invoke a simpler type of change to vSphere by requesting a snapshot to a VM.

Understanding Snapshots

Snapshots represent the state of a virtual machine at a particular point in time. A virtual machine can be reverted to a snapshot, undoing all changes made to its memory and disk since the snapshot was taken. You can take a snapshot only of the disk state; this type of snapshot is used when backing up virtual disks. VI supports multiple snapshots for a virtual machine and allows you to navigate to any snapshot. As each additional snapshot is taken, a tree of snapshots is formed. This introduces some complexity in writing scripts for snapshots: You must understand how to find the snapshot objects as well as how to manipulate them.

The following diagram shows the relationships of the key server-side objects and data objects:

- **VirtualMachine** is the server-side object that represents a virtual machine. It has a large number of properties and methods, but only two are of interest for managing snapshots. The first property, snapshot, allows you to access the snapshot information for the virtual machine. However, if you want to manipulate the snapshot files, you need to use the layout property to navigate to the snapshots. VirtualMachine also has two useful methods:
  - CreateSnapshot
  - RemoveAllSnapshots
- **VirtualMachineSnapshotInfo** is a data object. It is the root of the snapshot tree and identifies which snapshot is considered the current snapshot. The rootSnapshotList is an array of VirtualMachineSnapshotTree objects that represent the top-level snapshots in the tree.
- **VirtualMachineSnapshotTree** is a data object. It contains much of the useful metadata about a snapshot, such as its creation time, description, and name. It also contains a list of the child snapshots of this snapshot, if any child snapshots exist. The VirtualMachineSnapshotTree object always has a reference to the actual server-side snapshot object.
- **VirtualMachineSnapshot** is the server-side object that represents a snapshot. It contains some configuration info for the virtual machine, which was configured at the time the snapshot was taken. VirtualMachineSnapshot supports the following methods:
  - RemoveSnapshot
  - RenameSnapshot
  - RevertToSnapshot
- **VirtualMachineFileLayout** is a data object that contains information about the various files that the virtual machine uses. For snapshots, the only relevant property is named snapshot; it provides links to the VirtualMachineSnapshotLayout objects for this virtual machine.
- There is one **VirtualMachineFileLayoutSnapshot Layout** data object for each snapshot; it contains a link to the VirtualMachineSnapshot object and a list of all the files that comprise the snapshot.



[A complete list of properties for each object can be found in the VI SDK Reference Guide *(http://www.vmware.com/support/developer/vc-sdk/visdk25pubs/ReferenceGuide/index.html).]*

Track the reconfiguration task progress

Once the snapshot creation task is created, it's status can be determined by using the PropertyCollector. A filter is created for the info.state property of Tasks using the property collector. The WaitForUpdates method of the service connection will then return any updates to the filtered properties. Exercise 3 covered WaitForUpdates and property filters. In this case, you'll be monitoring the state of a task object.

1.2 Look up the task object in the Reference Guide and note which property stores the status.

Since *info.state* of a task indicates the current task state (error, queued, running, success), waiting for changes here will alert to any state changes.

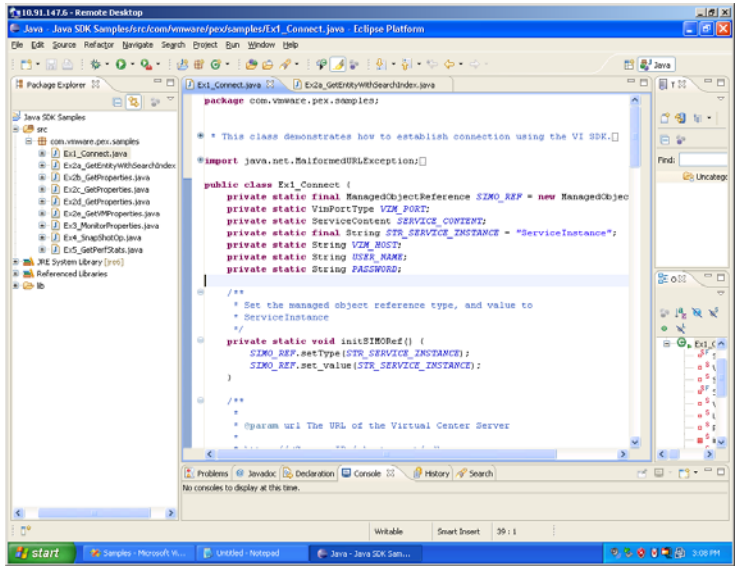
Step 2: Open the lab files for Ex4_SnapshotOps.java

2.1 If you have not already opened eclipse, begin by opening “Java Lab” on the desktop.

Your IDE should be setup for you as shown below.

2.2 The source code is set for you on the left window in the Package Explorer under *src / com.vmware.pex.samples*. Double click the icon on the left-hand panel that corresponds to the code for this exercise number (the solutions for each exercises are also listed). Just selecting it may not work as it will only change your working file if you already have it open.

2.3 You should browse the code in the main window. Please do not compile or run the code until you get to that section of the instructions.



Step 3: Functional Walk-through for Ex4_SnapshotOps.java

- initAll ()
    - initSIMORef ()
    - initVimPort ()
        - locator.setMaintainSession(true);
        - VIM_PORT = locator.getVimPort(new URL(url));

  - initServiceContent ()
    - SERVICE_CONTENT = VIM_PORT.retrieveServiceContent(SIMO_REF);
  - connect ()
    - VIM_PORT.login(SERVICE_CONTENT.getSessionManager(), uname, pword, null);
  - initPropertyCollector ()
    - PROP_COLLECTOR = SERVICE_CONTENT.getPropertyCollector();
  - initRootFolder
    - ROOT_FOLDER = SERVICE_CONTENT.getRootFolder();

- **createSnapShot(vmName, snapshotName);**
  This is a function you will be writing or pasting for the lab. It is currently blank and the code is covered below.
  createSnapShot creates a snapshot with the specified name on the VM of the specified name. Because this is a reconfiguration of a managed object, the action takes place on the server and is tracked by a task.
  - cssTask = VIM_PORT.createSnapshot_Task(getVMByName(vmName), ssName, "Testing create snap shot", false, true);
    This line calls createSnapshot_Task, a VI SDK method that acts on a VM managed object. The Reference Guide explains the arguments to use here. The first is the MORef of the VM (taken indirectly from the command line), the second is the name for the snapshot (taken indirectly from the command line), the third is the description for the snapshot, the fourth indicates whether to include the memory state of the VM in the snapshot (if this is false, the snapshotted VM must start powered off), the fifth argument indicates whether to quiesce the file system.
  - waitForValues( )
    waitForValues watches the the managed object in the first argument (in this case the newly created task) and gathers the properties listed in the second argument (in this case info.state, info.error, and info.progress). It stops watching when the property(ies) in the 3rd argument match the criteria in the 4th argument (in this case, status is success or error). The structure of tasks is explained in the reference guide.
    The waitForValues function is essentially a more general version of the monitorProperties function in exercise 3. The portion of the code that monitors properties is essentially unchanged. The major difference is in this version the ending conditions are settable.
    - createPropertyFilter(propArr);
      createPropertyFilter takes the name of a property to monitor (this is indirectly from the command line) and creates a propertyFilter that will flag changes to that property in any VM on the server.
      The code to do this is nearly the same as the code to get the property in the first place. Compare this with GetVMInfo () in GetProperties.java. The only difference is that the VI SDK method used is createFilter

instead of retrieveProperties. Essentially, the code here creates a PropertyFilterSpec that traverses the inventory to cover all VMs and selects the specified property.

- getVMTraversalSpec()
  This is the same function used in a number of the previous exercises to traverse the VMs.
- VIM_PORT.createFilter(PROP_COLLECTOR, spec, true);
  This is essentially the exact same filter creation as in the previous exercise. The only differences here is that spec changed to reflect monitoring a task, and the last argument is now true, indicating that partial changes should only returned to changed portion.

- VIM_PORT.waitForUpdates(PROP_COLLECTOR, version);
  The process of waiting for updates is also exactly the same as the previous exercise.
- updateset.getVersion();
  As before, the version number is verified.
- updateset.getFilterSet();
  As in the previous exercise, the filterset is unwrapped in a series of loops, once it is accessed in this step.
- for (int fi = 0; fi < filtupary.length; fi++) {
  As in the previous exercise, this line begins a loop iterating through the propertyFilterUpdates. Each propertyFilterUpdate represents a specific set of managed object updates. The syntax here is a little different (due to programmer preference), but the effect is exactly the same as the line for " for (PropertyFilterUpdate pfu : pfuArr)" in the previous exercise.
  - objupary = filtup.getObjectSet();
    As with the previous exercise, the ObjectSet of each propertyfilter represents the array of each particular object that had updates for that set. Once again, the syntax here is a bit different.
  - for (int oi = 0; oi < objupary.length; oi++) {
    Exactly like the previous exercises, this line iterates through the objectSet, going through one changed object for each loop. The syntax here is a bit different for stylistic reasons.
    - PropertyChange[] pcArr = ou.getChangeSet();
      Exactly as before, for each object, any number of property changes could have occurred. This gets an array indicating the set of property changes. The array is iterated via a for loop so that the values can be compared to the requested values to end the wait.
      - for (int ci = 0; ci < propchgary.length; ci++) {
        This line increments through each of the changed properties. This is a change from the previous exercise to allow this function to keep track of end conditions.
        - updateValues(endWaitProps, endVals, propchg);

For each property listed in the first argument, updateValues updates the array in the second argument with the property change in the third argument. It also leaves a property blank if it has been deleted. The result is that the endVals array has the latest properties for everything that is being watched for end conditions.

- updateValues(filterProps, filterVals, propchg);
  Another call to updateValues handles the same thing, but in this case makes sure the filterVals array is kept up to date for all the properties that are being watched by the filter.

- for (int chgi = 0; chgi < endVals.length && !reached; chgi++) {
- for (int vali = 0; vali < expectedVals[chgi].length && !reached; vali++) {
  These two lines set up a double loop to determine if the expected values have been reached and exit the outer loop (and stop waiting) if they have.
- VIM_PORT.destroyPropertyFilter(filterSpecRef);
  This line simply destroys the property filter now that its work is done.

- getVMSnapshotFromTask(cssTask);
  getVMSnapshotFromTask is identical to the earlier functions like getVMInfo from exercise 2a, except it uses a Task object and looks specifically for the info.result property.

- removeSnapShot(VM_SNAPSHOT, true);
  removeSnapShot takes the MORef of a snapshot as the first argument and deletes it. The snapshot is passed to it by the command line or through the createSnapshot function. The second argument signals whether or not to delete child snapshots.
  - rssTask = VIM_PORT.removeSnapshot_Task(vmss, removeChildren);
    This line calls the webservice and removes a snapshot by creating a task. Tracking the task uses the waitForValues method described above.
  - waitForValues()
    This is the same function as described above.
- removeAllSnapShots(vmName);
  removeSnapShot takes the name of a VM as the first argument and removes all of its snapshots. The VMname is passed to it indirectly through the command line.
  - rassTask = VIM_PORT.removeAllSnapshots_Task(getVMByName(vmName));
    This line calls the webservice and removes a snapshot by creating a task. Tracking the task uses the waitForValues method described above. Note that embedded in the function call is a call to getVMByName, the

same function that has been used several times in this lab to get the MORef of a VM using the property collector.

- waitForValues();
  This is the same function as described above.

Step 4: Modify Code

In this lab, there are several steps you'll be providing. In order to create a snapshot, the first step is to run the createSnapshot_Task on the appropriate VM. The result is a task. Next, a filter needs to be setup and monitored to check when the task is complete. This is accomplished by using a function called waitForValues, which is just a value checker on the end of the monitorProperties function you created in the previous exercise. Finally, when the end state of the task is returned, the function indicates success or failure.

4.1 Fill-in the createSnapShot function as follows:

```
    public static void createSnapShot(String vmName, String ssName) {
    try {
// Begin fill-in
// v v v v v v v

        ManagedObjectReference cssTask = VIM_PORT.createSnapshot_Task(
               getVMByName(vmName), ssName, "Testing create snap shot",
               false, true);
        Object[] result = waitForValues(cssTask, new String[] {
               "info.state", "info.error", "info.progress" },
               new String[] { "state" }, // info has a property - state for
               // state of the task
               new Object[][] { new Object[] { TaskInfoState.success,
                  TaskInfoState.error } });
        if (result[0].equals(TaskInfoState.success)) {
            System.out.println("Success: SnapShot creation");
            VM_SNAPSHOT = getVMSnapshotFromTask(cssTask);
        } else {
            System.out.println("Failure: SnapShot creation");
        }
// ^ ^ ^ ^ ^ ^ ^
// End fill-in

    } catch (Exception e) {
```

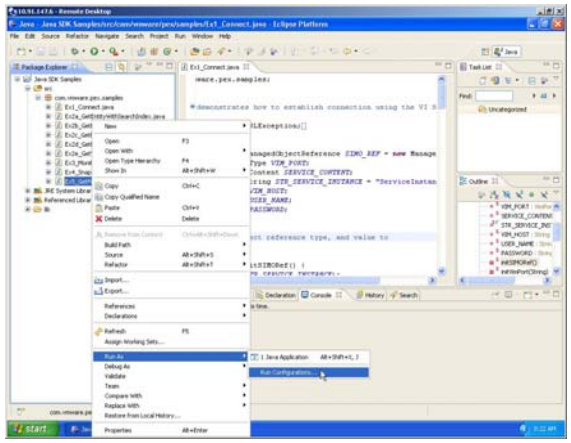
```
            e.printStackTrace();
        }
    }
```

Step 5: Compiling and Running the Code

Once you've made the necessary changes to the code, compile and run the code. Despite your normal preferences when running and compiling code, please use the procedure in the lab to make it as easy as possible for the lab staff to diagnose any problems you have.

5.1 Verify that the program is complete and that you have filled in all the code sections specified in step 5.

5.2 Right click on the file that you were working on in the Package Explorer, go to “Run As” and click “Run Configurations”. A new window will open up with a space for providing the arguments to the command. Recall that a number of the variables in the program are taken from the command line options.

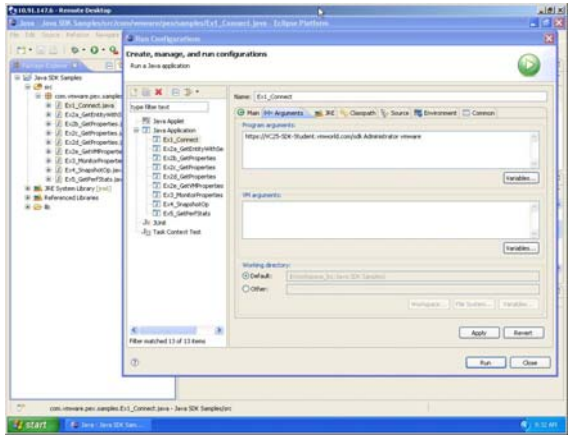


5.3 On the right (Run Configuration) window, confirm up top that it’s the correct file and click on the “Arguments” tab. This will also be the file that is highlighted in the pane on the left side of the Run Configuration window. If you do not have the correct file selected, you will re-run the last program.Input the arguments that that this program requires. You will need to input the following:

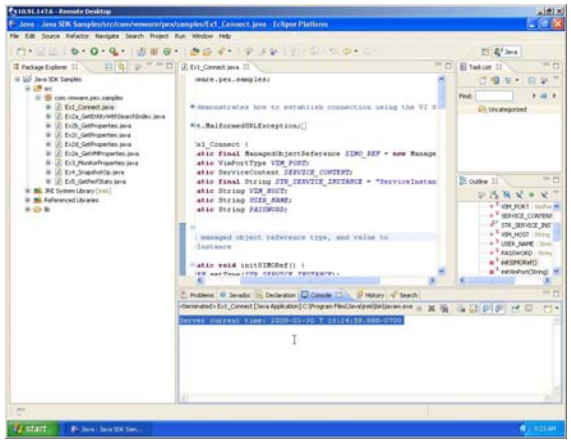
1) Virtual Center host (the VIM_HOST listed at your station)
2) Username (the USERNAME listed at your station)

3) Password (the PASSWORD listed at your station)
4) VM name (the MY_VM_NAME listed at your station) - *the name of the VM that you want to snapshot*
5) create snapshot1 – *desired operation*

Usage: https://<VIM_HOST>/sdk <USERNAME> <PASSWORD> <MY_VM_NAME> create snapshot1
Example : https://vc01-student01.vmworld.com/sdk sdk-student01 abc123 winsvc create snapshot1



5.4 Click Apply and Run Application at the bottom of the screen. You should see the correct output in the Console tab just below your code.



5.5 After the snapshot is created, press "r" in the console window and press enter to remove the snapshot.

Optional - Hands-on Exercise 5: Getting performance data

The goal of this exercise is to facilitate an understanding of the performance data that is available in VirtualCenter and VMware ESX hosts and how to collect and manipulate this data. Both VirtualCenter and VMware ESX hosts maintain performance data. A VMware ESX server maintains performance data only about itself and the virtual machines and resource pools that exist on that host. VirtualCenter maintains performance data for all its clusters, hosts, resource pools, and virtual machines. Performance data is collected at multiple defined intervals. The information is collected on both the host and the VirtualCenter system for a 20-second sample interval and a 5-minute interval, retained for 1 hour and 1 day, respectively. VirtualCenter maintains historical performance data for the 30 minute sample interval, 2-hour sample interval, and 1-day sample interval, which are retained for 1 week, 1 month, and 1 year, respectively.

Step 1: Understanding performance counters and metrics

You must understand a few key points about collecting performance information in VMware Infrastructure. The first point is that performance information is not part of the normal properties of an object; rather performance information is accessed through a special managed object in the VI API. This object is called PerformanceManager.

The second point is the difference between counters and values. A counter indicates a type of collected performance information. Counters represent a description of how some performance measure is recorded. A counter does not provide any information about the actual performance data or the object on which the data was collected. The counter only describes a type of data that could be collected. Counters store the following:

- Type of units (e.g., percent, milliseconds, KB)
- Text description of what the counter measures
- Statistic type (amount of change, point-in-time value, or rate)
- How the information is rolled up over longer periods (average, minimum, maximum, summation over time, of the latest value)
- Minimum VirtualCenter log level at which the statistic is recorded

Counters are divided into groups according to categories (such as cpu, disk, memory), and are usually represented by a dotted string notation, which includes the counter's group, name, and roll-up type. For example, cpu.usage.min indicates the minimum CPU usage in a collected sample. A list of all VMware Infrastructure 3 performance counters is in Appendix B of the *VI SDK Programming Guide* (in vSphere 4, this information has moved to links off the main page for the Performance Manager)

By contrast, a performance value represents the actual information collected, rather than the description of that

information.

For the entities, such as a host or virtual machine, there may be multiple instances of the device for which the same performance counter can be collected. Each instance is a performance metric. For example, Usage is a performance counter for CPU utilization, while Usage collected for CPU #1 for a host is a performance metric, which is represented as counter cpu usage with device instance 1. A performance metric is identified by a MetricId. The MetricId consists of two parts:

- *CounterId*: The integer that identifies the performance counter.
- *InstanceId*: The name of the instance such as CPU "0" or disk "A".

The Metric also contains the values of the performance interval—this is the actual performance information.

Understanding the Performance Manager

PerformanceManager is a managed object that provides an interface for querying performance statistics from the VI3 environment. Any managed objects that have performance statistics available are called *performance providers*. Like all managed objects (server-side objects) in the VI SDK, the first step to using the PerformanceManager is locating it (since this is a non-inventory object, it is off the ServiceInstance) and determining its MORef.

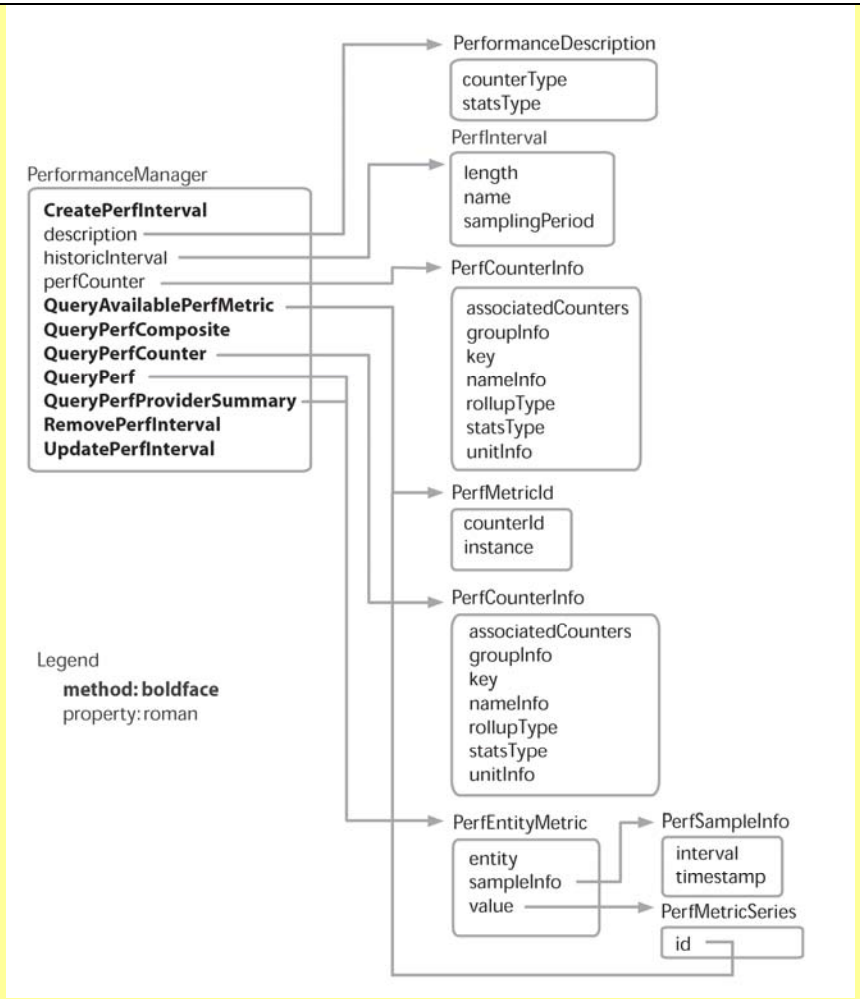
The PerformanceManager has properties that define the scope of all the performance counters on the system. For instance, the PerformanceManager contains the definitions of all the performance counters. Using the PerformanceManager “holder” for the definitions allows a properly constructed program to become aware immediately of any new performance metrics, without having to be rewritten for new data structures.

One very important thing to note is that the PerformanceManager does *not* contain actual performance results. Those can be accessed from various methods the PerformanceManager provides, but are not directly available as properties.

The PerformanceManager provides methods for accessing metrics, counters, and performance values separately:

- Finding the available performance metrics usually uses the QueryAvailablePerfMetric ( ) or QueryPerfProviderSummary ( ) method.
- Finding the counter metadata information describing what those metrics represent can be accomplished several ways. If the appropriate Metric is already located, the counter it associates with is simply a property of the metric. To locate counters when the metric is unknown, this information can be found using the QueryPerfCounter ( ) method or it is also possible to get the information on all counters directly via the perfCounter property of the PerformanceManager.
- Finding the actual performance statistics for those metrics uses the QueryPerf ( ) method.

The diagram below shows the structure of the PerformanceManager:



Understanding how to find counters and metrics

To determine the metrics you need, you will need to get a list of the defined PerfMetrics for the host. The PerformanceManager provides the QueryAvailablePerfMetric in order to get the metrics list for a given server-side object.

Once the metrics are found using QueryAvailablePerfMetric, they need to each be associated with the appropriate counter. Counters are stored in the perfCounter property of the PerformanceManager and need to be mapped to the counter IDs used in the metrics by use of a loop.

Understanding how to query the performance statistics

In order to query performance statistics, it is necessary to create a PerfQuerySpec. A PerfQuerySpec is built with the following data:

- ManagedObjectReference of the object being analyzed
- The format to return the statistics in (an array or a CSV list)
- The interval ID (referring to a default in exercise 3)
- The maximum number of samples to be returned
- The begin time
- The end time
- The list of PerfMetricID names to return (the array from the previous step)

Once the PerfQuerySpec is created, the PropertyCollector's QueryPerf method can be run to get the performance data. Running the QueryPerf method returns a PerfEntityMetric object (or a CSV version if different arguments are specified). A PerfEntityMetric consists of three types of information:

- the entity being collected from (entity)
- the data points collected (value, a PerfMetricSeries array)
- the time represented by each datapoint (*sampleInfo*, a perfSampleInfo array, where each perfSampleInfo object consists of a timestamp and an interval length).

In order to output this data in exercise 3, a loop steps through each value of the PerfMetricSeries array, which will only contain the metrics specified above, outputting the appropriate counter and metric information.

In exercise 3, the entire PerformanceManager process is performed twice—once to output the information about the VMware ESX server and once to output the information about each virtual machine.

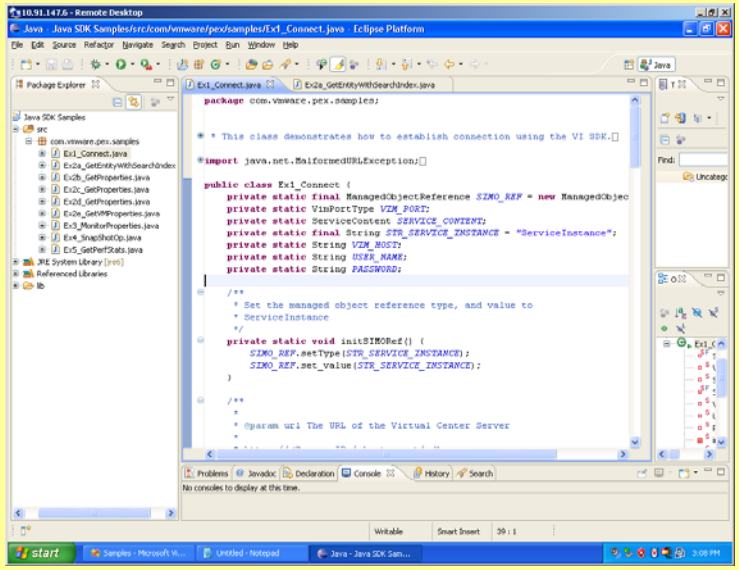
Step 2: Open the lab files for Ex5_GetPerfStats.java

2.1 If you have not already opened eclipse, begin by opening "Java Lab" on the desktop.

Your IDE should be setup for you as shown below.

2.2 The source code is set for you on the left window in the Package Explorer under *src / com.vmware.pex.samples*. Double click the icon on the left-hand panel that corresponds to the code for this exercise number (the solutions for each exercises are also listed). Just selecting it may not work as it will only change your working file if you already have it open.

2.3 You should browse the code in the main window. Please do not compile or run the code until you get to that section of the instructions.



Step 3: Functional Walk-through for Ex5_GetPerfStats.java

- initAll ()
  - initSIMORef ()
  - initVimPort ()
    - locator.setMaintainSession(true);
    - VIM_PORT = locator.getVimPort(new URL(url));
  - initServiceContent ()
    - SERVICE_CONTENT = VIM_PORT.retrieveServiceContent(SIMO_REF);

  - connect ()
    - VIM_PORT.login(SERVICE_CONTENT.getSessionManager(), uname, pword, null);
  - initPropertyCollector ()
    - PROP_COLLECTOR = SERVICE_CONTENT.getPropertyCollector();
  - initRootFolder
    - ROOT_FOLDER = SERVICE_CONTENT.getRootFolder();
  - initPerfMgr
    Because performance statistics are not stored with the configuration data, the
    - ROOT_FOLDER = SERVICE_CONTENT.getPerfMgr();

- getPerfCounters();
  getPerfCounters lists all the valid performance counters for the system it is connected to. At a high level, this function simply uses the propertycollector to get all the perfCounters in the performance manager, then stores them in a hash.
  - oCont = VIM_PORT.retrieveProperties(PROP_COLLECTOR, propertyFilterSpecs);
    This line makes a call to retrieve properties to get all the performance counters. The section of code following this example builds a hash table out of the metrics. A hash table is an array that is setup to allow key-value searches.
- getAvailablePerfMetricIds(entityName, isVM);
  - getVMByName(entityName);
    This function simply gets the MORef of a VM based on the name. The name here comes indirectly from the command line.
  - getHostByName(entityName);
    This function is triggered if the command line is not a VM. It gets the MORef of a host based on the name. The name here comes indirectly from the command line.
  - VIM_PORT.queryAvailablePerfMetric(PERF_MGR, entityMor, null, null, new Integer(20));
    This line makes a call to the webservice to get all the performance metrics available for the specified object. As with all VI API calls, the first argument is the MORef of the object the call relates to, in this case the performance manager. The second argument is the entity to get the metrics for (this is indirectly from the command line). The third and fourth arguments are the begin and end time; since both are null here, all metrics in the system are collected. The fifth argument is the interval ID; in this case the 20 second (shortest) interval is specified.
- getPerfStats(entityName, isVM, perfCounter);
  getPerfStats retrieves the performance statistics from the server. It returns the requested (3rd argument) stats for the specified object (first argument). The second argument specified whether the object is a VM (otherwise it is assumed

to be a host).

- **getPerfQuerySpec(entityName, isVM, perfCounter)**
  This is a function you will be writing or pasting for the lab. It is currently blank and the code is covered below.
  getPerfQuerySpec constructs a perfQuerySpec object to be used with the queryPerf VI API method.
  A perfQuerySpec contains the intervalID and maximum number of samples to retrieve, as well as the metric ID. The metric ID contains a counter ID and an instance ID. The instance ID covers cases where there may be multiple instances of the same counter.
- Integer counterInteger = counters.get(perfCounter);
  This line uses the Java hash functionality. It looks up a counter ID based on the counter name. This uses the counter hash table built in getPerfCounters. Without that hash table, determining the counter ID of a specific counter name requires iterating through all the counters. Because most programs using counters will need to do this frequently, storing the information in a hash or other lookup scheme is a good idea. Another important point is that if the VC or ESX server reboots, the counter names could change. Because of this, it is best to rebuild the hash if the connection is lost.
- retVal = VIM_PORT.queryPerf(PERF_MGR, new PerfQuerySpec[]{ ---see line above ---});
  This line calls the queryPerf method of the VI API and obtains the performance metrics for the specified counter. The format is covered in the VI SDK Reference Guide. Like all VI API calls, the first argument is the MORef of the associated managed object, in this case the performance manager. The second argument is the perfQuerySpec that specifies what metric to get.
- disconnect();

Step 4: Modify Code

The code sample for this exercise concentrates on how to create a PerfQuerySpec. Before doing so, the program needs to map the counter IDs to counter names so that the right data can be requested. This is a bit complex and is covered in the getPerfCounters() function. In this code, the counters are already accessible by name in the form of a hash. For folks unfamiliar with hashes, the "counters.get(perfCounter)" code simply retrieves the item in the counters hash with a name matching perfCounter.

The rest of this code creates the PerfQuerySpec. It contains several bits of information:

The interval length to retrieve over (in seconds)

The maximum number of samples (you can alternatively specify begin and end times)

The PerfMetricID (a datastructure that contains the counter ID and instance)

The getPerfQuerySpec that gets returned by this function is later used as an argument to the queryPerf() method, which returns the performance metrics.

4.1 Fill-in the getPerfQuerySpec method as shown below:

```
     private static PerfQuerySpec getPerfQuerySpec(String entityName,
        boolean isVM, String perfCounter) {
     PerfQuerySpec retVal = null;

     try {
// Begin fill-in
// v v v v v v v
        Integer counterInteger = counters.get(perfCounter);
        if (counterInteger == null) {
           System.out.println("Counter doesn't exist..........");
           System.exit(1);
        }
        int counterId = counterInteger.intValue();
        System.out.println("Counter id " + counterId+ " - "+perfCounter);

        if (!metrics.containsKey(counterInteger)) {
           System.out
                  .println("This Counter is not supported in this entity $$$$$$$$$$$$$$$$$$");
           System.exit(1);
        }

        PerfMetricId pmid = new PerfMetricId();
        pmid.setInstance("*");
        pmid.setCounterId(counterId);

        PerfQuerySpec pqSpec = new PerfQuerySpec();
        // You can obtain this from the PerfProviderSummary.
        // By default 20 sec interval exists.
        // When connecting to ESX host directly, interval value 0 can be
        // used.
        // But the value retrieved will be the same in that 20 second
        // window.
        pqSpec.setIntervalId(new Integer(20));
        pqSpec.setMaxSample(new Integer(1));
        pqSpec.setMetricId(new PerfMetricId[] { pmid });
```

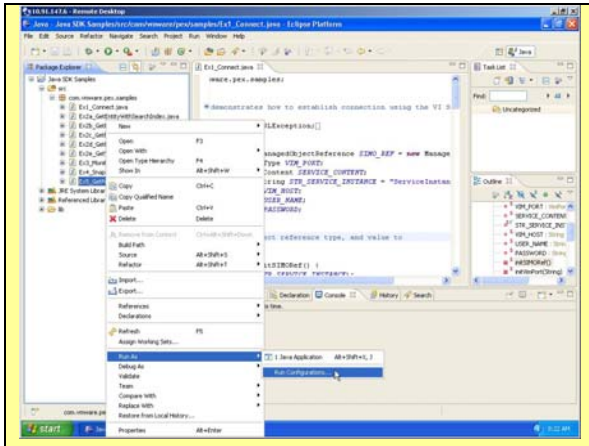
```
        ManagedObjectReference entityMor = null;
        if (isVM) {
           entityMor = getVMByName(entityName);
        } else {
           entityMor = getHostByName(entityName);
        }
        pqSpec.setEntity(entityMor);

        retVal = pqSpec;
// ^ ^ ^ ^ ^ ^ ^
// End fill-in
     } catch (Exception e) {
        e.printStackTrace();
     }

     return retVal;
  }
```

Step 5: Compiling and Running the Code

Once you've made the necessary changes to the code, compile and run the code. Despite your normal preferences when running and compiling code, please use the procedure in the lab to make it as easy as possible for the lab staff to diagnose any problems you have.

5.1 Verify that the program is complete and that you have filled in all the code sections specified in step 5.

5.2 Right click on the file that you were working on in the Package Explorer, go to “Run As” and click “Run Configurations”. A new window will open up with a space for providing the arguments to the command. Recall that a number of the variables in the program are taken from the command line options.

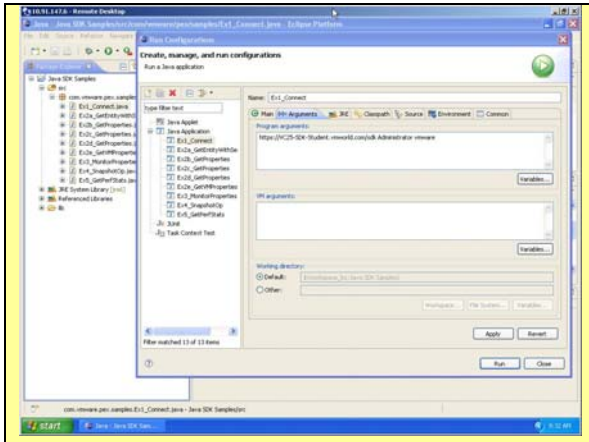


5.3 On the right (Run Configuration) window, confirm up top that it's the correct file and click on the "Arguments" tab. This will also be the file that is highlighted in the pane on the left side of the Run Configuration window. If you do not have the correct file selected, you will re-run the last program.

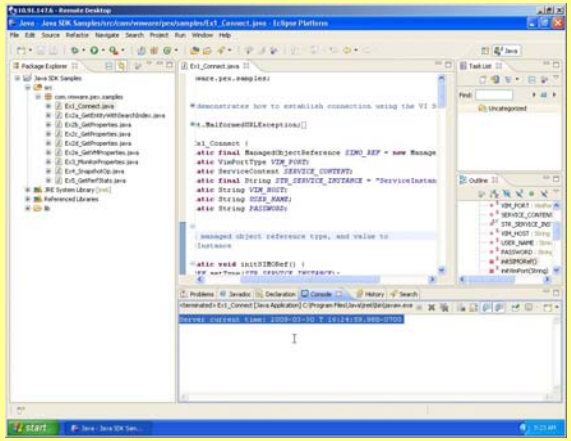
Input the arguments that that this program requires. You will need to input the following:

1) Virtual Center host (the VIM_HOST listed at your station)
2) Username (the USERNAME listed at your station)
3) Password (the PASSWORD listed at your station)
4) VM name (the MY_VM_NAME listed at your station) - *the name of the VM that you want to stats from*
5) true – *Indicates TRUE if the target is a VM*
6) cpu.usagemhz.average – *name of the stat you want to get*

Usage: https://<VIM_HOST>/sdk <USERNAME> <PASSWORD> <MY_VM_NAME> TRUE cpu.usagemhz.average
Example : https://vc01-student01.vmworld.com/sdk sdk-student01 abc123 winsvc TRUE cpu.usagemhz.average



5.4 Click Apply and Run Application at the bottom of the screen. You should see the correct output in the Console tab just below your code.



Appendix A: Lab setup

The lab we used consisted of a number of ESX servers managed by the same VC server. The VC server managed a number of VMs, some configured to be read-only to the users and others configured to be read/write by the users. The VC should have a DRS cluster configured.

While the names used for many of the VMs are not important, some exercises find VMs based on particular criteria.

Each student should have the following:

1. A folder called Student<ID>
2. A resource pool called Student<ID>
3. The folder Student<ID> should have a VM called Student01_VM1. You can use another VM, as long as <VM> points to it. This VM should allow <login> to have "VMworld rw" permissions.
4. A virtual port group called "Production_LAN". This can be internal only.
5. A resource pool called "Admin RP"
6. The Admin RP resource pool should have a VM called vmw9-1-winsrv01. You can use a different name, as long as <vm-ro> points to it.
7. A role called "VMworld global". It should be a copy of the "read-only" role with the following additional permissions:
    a. Global:Diagnostics
    b. VM:provisioning:customize
    c. VM:provisioning:deploy template
    d. VM:provisioning:read customization spec
    e. VM:provisioning:modify customization spec
    f. Datastore:Browse datastore
8. A role called "VMworld rw". It should have the following permissions:
    a. Global: Log Event
    b. Global: Diagnostics
    c. Datastore:Browse datastore

   d. Datastore:File management
   e. Host:Configuration:Local Operations:Create VM
   f. Host:Configuration:Local Operations:Delete VM
   g. Virtual Machine (all check boxes)
   h. Resource:Assign VM to Resource Pool
   i. Resource:Apply Recommendation
   j. Resource:Migrate
   k. Resource:Relocate
   l. Resource:Query Vmotion
9. A login called l09-student01. You can use a different login, but set the <login> variable appropriately.
10. Add "VMware Global" permission for <login> for all users at the hosts & clusters level (above the datacenter)
11. For the <login> user, remove "VMworld rw" permissions from everywhere but the student resource pools and folders.
12. Add the "VMworld rw" permission to each student resource pools and folders for the corresponding <login>
13. Create a new template. It should be a clone of a windows VM.
14. Create a VM called vmw9-1-winsrvVM-DEVas a clone of the template. Give it 2 vCPUs and connect it to the "Production_LAN" port group. Put it in the "Admin RP" resource pool.

For large lab environments, where many students will be trying this lab on the same VC server at the same time (and only in many-student environments), there may be an issue with the number of open SDK sessions. This is generally not an issue with "normal" production environments, but large numbers of users opening (and possibly never closing) SDK connections over the course of a lab like this could open a lot of connections. For instance if students are doing all the exercises and requiring 4 attempts for each, this may generate 80 or more connections per user over the course of the lab.

To avoid this issue, classroom environments with a "shared" VC approach may want to make the following change to increase the number of allowed SDK sessions in VC:

Make the following addition (see in red below) to the vpxd.conf file (Located in C:\Documents and Settings\All Users\Application Data\VMware\VMware VirtualCenter) on the Virtual Center Server, then restart the Virtual Center service.

```
<config>
        <!-- <ws1x> -->
                <!-- Enables 1.x Web Services compatibility -->
                <!-- <enabled>true</enabled> -->
                <!-- Enables 1.x Built in Perf data Refresh -->
                <!-- <perfRefreshEnabled>false</perfRefreshEnabled> -->
                <!-- Location of ws1x persistent data. Default is same directory as vpxd.cfg -->
                <!-- <dataFile> C:\ws1x.xml </dataFile> -->
                <!-- Location of the ws1x event mapping file. Default is the same directory as vpxd.cfg -->
                <!-- <eventMap> C:\ws1xEventMap.xml </eventMap> -->
        <!-- </ws1x> -->

        <vpxd>
                <das>
                        <serializeadds>true</serializeadds>
                        <slotMemMinMB>256</slotMemMinMB>
                        <slotCpuMinMHz>256</slotCpuMinMHz>
                </das>
                <filterOverheadLimitIssues>true</filterOverheadLimitIssues>
        </vpxd>
        <vmacore>
                <threadPool>
```

```
                <TaskMax>30</TaskMax>
            </threadPool>
            <soap>
                <maxSessionCount>1000</maxSessionCount>
            </soap>
        </vmacore>
</config>
```