CLOUDSIM 3.0.3 MANUAL

A Researcher's Guide to Simulation

Anupinder Singh
CloudsimTutorials.online

Preface

CloudSim 3.0.3 manual is an effort to make available a ready to use document for researchers, working on advancement of cloud computing. This document is created by intensively analyzing the CloudSim source code and with reference of a relevant paper titled “CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms” published by Rajkumar Buyya & his team. This paper was published in 2011 under ‘Software: Practice and Experience’ journal of ‘Wiley Online Library’.

Index

1. CloudSim – Introduction & Installation

1. Background

Cloud computing is a pay as you use model, which delivers infrastructure (IaaS), platform (PaaS) and software (SaaS) as services to users as per their requirements. Cloud computing exposes data centers capabilities as network virtual services, which may set of required hardware, application with support of database as well as user interface. This allows the the users to deploy and access application across the internet which is based on demand and QoS requirements. As Cloud computing is a new concept and is still in a very early stage of its evolutions, so researchers and system developers are working on improving the technology to deliver better on processing, quality & cost parameters but most of the research is focused on quantifying the performance of provisioning policies. And to test such research on real cloud environment like Amazon EC2, Microsoft Azure, Google App Engine for different applications models under transient conditions is extremely challenging as:

1. Clouds exhibit varying demands, supply patterns, system sizes, and resources (hardware, software, and network).
2. Users have heterogeneous, dynamic, and competing QoS requirements.
3. Applications have varying performance, workload, and dynamic application scaling requirements.

Benchmarking the application performance on the real public cloud infrastructure like google cloud, Microsoft azure etc., is not suitable due to their multitenant nature as well as variable workload fulfillments. Also, there are very few admin level configurations that a user/researcher could be able to perform. Hence, this makes the reproduction of results that can be relied upon, an extremely difficult undertaking. Further, even if the we do, it is tedious and time consuming to re-configure benchmarking parameters across a massive-scale Cloud computing infrastructure over multiple test runs. therefore, it is impossible to undertake benchmarking experiments as repeatable, dependable, and scalable environments using real-world public Cloud systems.

Thus the need ot use of simulation tool(s) arises, which may become a viable alternative to evauate/benchmark the test workloads in a controlled and fully configurable environment which can reapatable over multiple iterations and reproduce the results for analysis. This simulation based approach various benefits across the researchers community as it allow them to:

1. Test services in repeatable and controllable environment.
2. Tuning the system bottlenecks (performance issues) before deploying on real clouds.
3. Evaluating different workload mix and resource performance scenarios on simulated infrastructures for developing and testing adaptive application provisioning techniques.

There are number of simulators that can be used to simulate the working of new services, and CloudSim is the more generalized and effective simulator for testing Cloud computing related hypothesis. CloudSim is an extensible simulation framework that allows seamless modeling, simulation and Experimentation relate to emerging cloud computing infrastructures and application services. Using CloudSim researchers and industry-based developers can test the performance of a newly developed

application service in a controlled and easy to set-up environment. Based on the evaluation results reported by CloudSim, they can further fine-tune the service performance.

2. Features of CloudSim

1. Support for modeling and simulation of large scale Cloud computing environments, including data centers, on a single physical computing node.
2. A self-contained platform for modeling Clouds, service brokers, provisioning, and allocation policies.
3. Support for simulation of network connections among the simulated system elements.
4. Facility for simulation of federated Cloud environment that inter-networks resources from both private and public domains, a feature critical for research studies related to Cloud-Bursts and automatic application scaling.
5. Availability of a virtualization engine that aids in the creation and management of multiple, independent, and co-hosted virtualized services on a data center node.
6. Flexibility to switch between space-shared and time-shared allocation of processing cores to virtualized services. These compelling features of CloudSim would speed up the development of new application provisioning algorithms for Cloud computing.

3. CloudSim Architecture

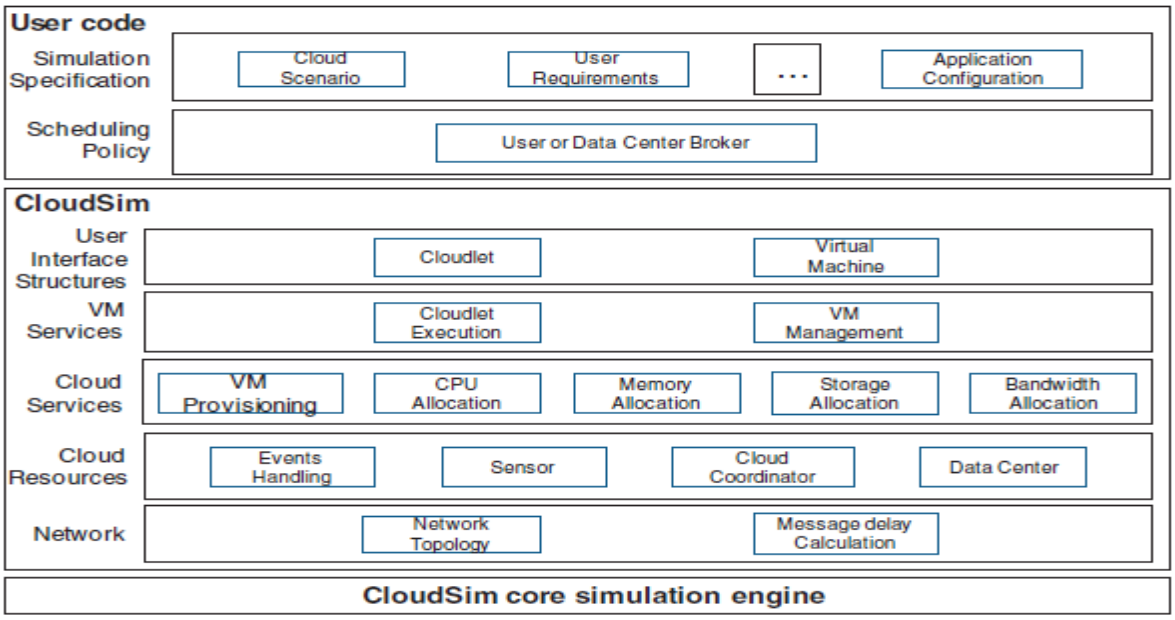


Figure 1: Layered CloudSim Architecture

Above diagram demonstrates about the layered architecture of CloudSim. The **CloudSim Core simulation engine** provides support for modeling and simulation of virtualized Cloud-based data center environments including queuing and processing of events, creation of cloud system entities (like data center, host, virtual machines, brokers, services etc.) communication between components and management of the simulation clock. The **CloudSim layer** provides the dedicated management interfaces for Virtual Machines, memory, storage, and bandwidth. Also it manages the other fundamental issues, such as provisioning of hosts to Virtual Machines, managing application execution, and monitoring dynamic system state(e.g. Network topology, sensors, storage characteristics etc) etc.

And the **User Code layer** is a custom layer where user write their own code to redefine the characteristics of the simulating environment as per their new research findings.

Design and Implementation of CloudSim

In this section, we provide finer details related to the fundamental classes of CloudSim, which are also the building blocks of the simulator. The overall class design diagram for CloudSim is shown in Figure 2.

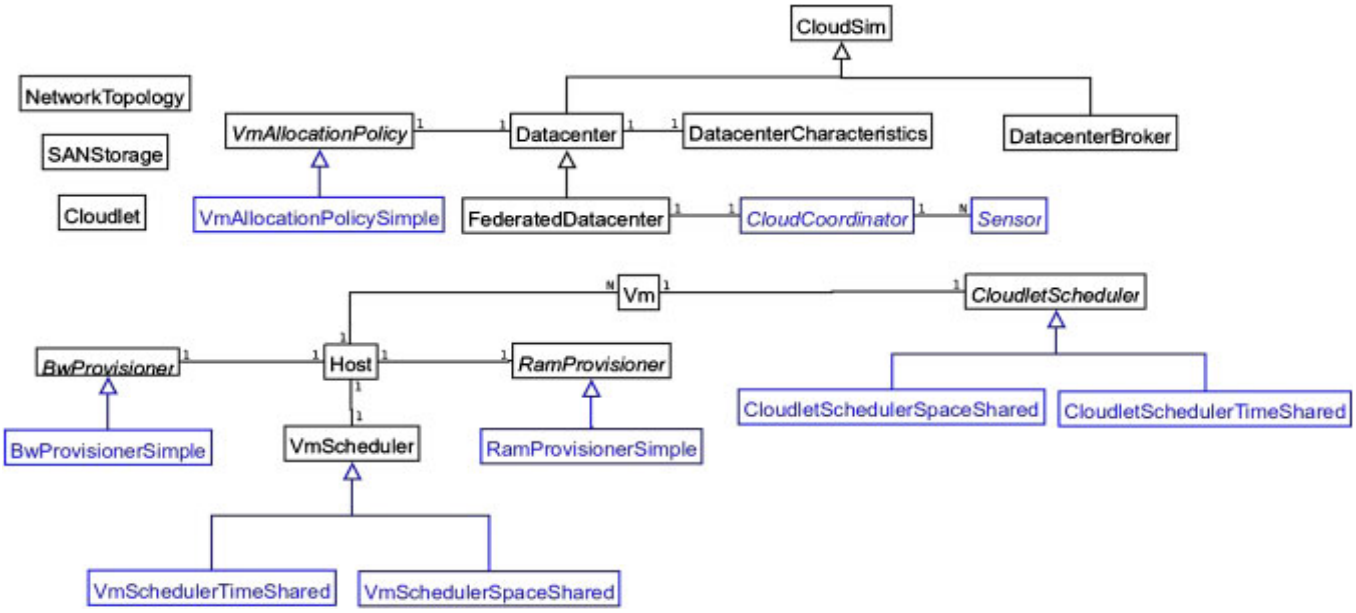


Figure 2: CloudSim Class Design Diagram

The description of all the major classes mentioned in class diagram above is described below:

1. **BwProvisioner**: The main role of this component is to undertake the allocation of network bandwidths to a set of competing VMs that are deployed across the data center. Cloud system developers and researchers can extend this class with their own policies (priority, QoS) to reflect the needs of their applications.
2. **CloudCoordinator**: It is responsible for periodically monitoring the internal state of data center resources and based on that it undertakes dynamic load-shredding decisions. Developers aiming to deploy their application services across multiple clouds can extend this class for implementing their custom inter-cloud provisioning policies.
3. **Cloudlet**: This class models the Cloud-based application services (program based tasks) such as content delivery, social networking, and business workflow. CloudSim mimics the complexity of an application in terms of its computational requirements. Every application service has a pre-assigned instruction length and data transfer (both pre and post fetches) overhead that it needs to undertake during its life cycle.
4. **CloudletScheduler**: This is responsible for implementation of different policies that determine the share of processing power among Cloudlets in a VM. There are two types of provisioning policies offered: space-shared (using **CloudetSchedulerSpaceShared** class) and time-shared (using **CloudletSchedulerTimeShared** class).
5. **Datacenter**: This class models the core infrastructure-level services (i.e. hardware) that are offered by Cloud providers (Amazon, Azure, and App Engine). It encapsulates a set of compute

hosts that can either be homogeneous or heterogeneous with respect to their hardware configurations (memory, cores, capacity, and storage). Also, every Datacenter component instantiates a generalized application provisioning component that implements a set of policies for allocating bandwidth, memory, and storage devices to hosts and VMs.

6. **DatacenterBroker or Cloud Broker**: This class models a broker, which is responsible for mediating negotiations between SaaS and Cloud providers; and such negotiations are driven by QoS requirements. The broker class acts on behalf of applications. It discovers suitable Cloud service providers by querying the CIS and undertakes online negotiations for allocation of resources/services that can meet the application's QoS needs. **Researchers and system developers** must extend this class for evaluating and testing custom brokering policies. The difference between the broker and the CloudCoordinator is that the broker represents the customer (i.e. decisions of these components are made in order to increase user-related performance metrics), whereas the CloudCoordinator acts on behalf of the data center, i.e. it tries to maximize the overall performance of the data center, without considering the needs of specific customers.
7. **DatacenterCharacteristics**: This class contains configuration information of data center resources.
8. **Host**: This class models a physical resource such as a compute or storage server. It encapsulates important information such as the amount of memory and storage, a list and type of processing cores (to represent a multi-core machine), an allocation of policy for sharing the processing power among VMs, and policies for provisioning memory and bandwidth to the VMs.
9. **NetworkTopology**: This class contains the information for inducing network behavior (latencies) in the simulation. It stores the topology information, which is generated using the BRITE topology generator.
10. **RamProvisioner**: This is an abstract class that represents the provisioning policy for allocating primary memory (RAM) to the Virtual Machines. The execution and deployment of VM on a host is feasible only if the RamProvisioner component approves that the host has the required amount of free memory. The RamProvisionerSimple does not enforce any limitation on the amount of memory that a VM may request. However, if the request is beyond the available memory capacity, then it is simply rejected.
11. **Vm**: This class models a Virtual Machine (VM), which is managed and hosted by a Cloud host component. Every VM component has access to a component that stores the following characteristics related to a VM (i.e.) accessible memory, processor, storage size, and the VM's internal provisioning policy that is extended from an abstract class called the CloudletScheduler.
12. **VmAllocationPolicy**: This abstract class represents a provisioning policy that a VM Monitor utilizes for allocating VMs to hosts. The main functionality of the VmAllocationPolicy is to select the available host in a data center that meets the memory, storage, and availability requirement for a VM deployment.
13. **VmScheduler**: This is an abstract class implemented by a Host component that models the policies (space-shared, time-shared) required for allocating processor cores to VMs. The functionalities of this class can easily be overridden to accommodate application-specific processor sharing policies.
14. **CloudSim**: This is the main class, which is responsible for managing event queues and controlling step-by-step (sequential) execution of simulation events. Every event that is generated by the CloudSim entity at run-time is stored in the queue called future events. These events are sorted by their time parameter and inserted into the queue. Next, the events that are scheduled at

each step of the simulation are removed from the future events queue and transferred to the deferred event queue. Following this, an event processing method is invoked for each entity, which chooses events from the deferred event queue and performs appropriate actions. Such an organization allows flexible management of simulation and provides the following powerful capabilities:

a. Deactivation (holding/pausing) of entities.
b. Context switching of entities between different states (e.g. waiting to active). Pausing and resuming the process of simulation.
c. Creation of new entities at run-time.
d. Aborting and restarting simulation at run-time.

15. **DeferredQueue**: This class implements the deferred event queue used by CloudSim.
16. **FutureQueue**: This class implements the future event queue accessed by CloudSim.
17. **CloudInformationService**: A CIS is an entity that provides resource registration, indexing, and discovering capabilities. CIS supports two basic primitives:
    a. **publish()**, which allows entities to register themselves with CIS and
    b. **search()**, which allows entities such as CloudCoordinator and Brokers in discovering status and endpoint contact address of other entities. This entity also notifies the other entities about the end of simulation.
18. **SimEntity**: This is an abstract class, which represents a simulation entity (such as Cloudlet, VM, Host etc) that is able to send messages to other entities and process received messages as well as fire and handle events. SimEntity class provides the ability to schedule new events and send messages to other entities, where network delay is calculated according to the BRITE model. Once created, entities automatically register with CIS. All entities must extend this class and override its three core methods:
    a. **startEntity**(), which define actions for entity initialization.
    b. **processEvent**(), which define actions processing of events.
    c. **shutdownEntity**(),which define actions entity destruction.

19. **CloudSimTags**. This class contains various static event/command tags that indicate the type of action that needs to be undertaken by CloudSim entities when they receive or send events.
20. **SimEvent**: This entity represents a simulation event that is passed between two or more entities. SimEvent stores the following information about an event:
    a. type,
    b. init time,
    c. time at which the event should occur,
    d. finish time,
    e. time at which the event should be delivered to its destination entity,
    f. IDs of the source and destination entities,
    g. tag of the event, and
    h. Data that have to be passed to the destination entity.
21. **CloudSimShutdown**: This is an entity class that waits for the termination of all end-user and broker entities, and then signals the end of simulation to CIS.

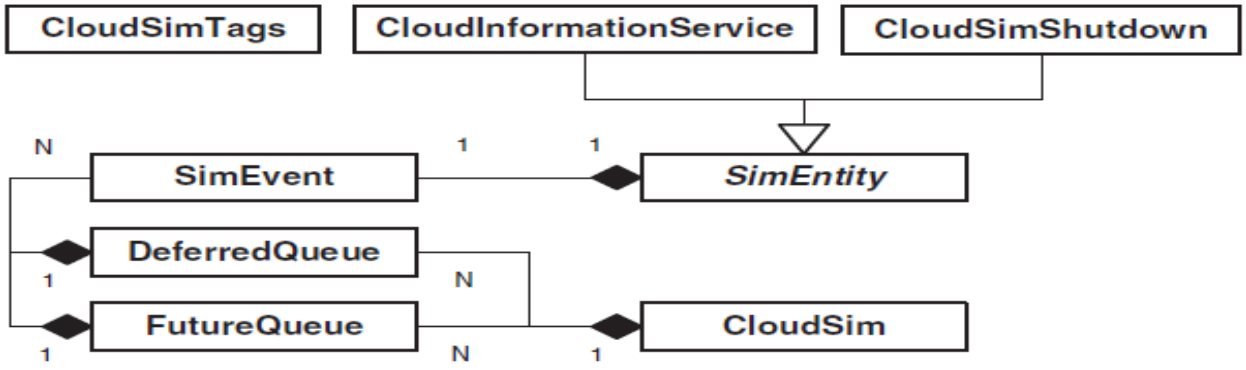


Figure 3: Interrelation of Core Simulation Classes

Following is the description of frequently used terms in this manual:

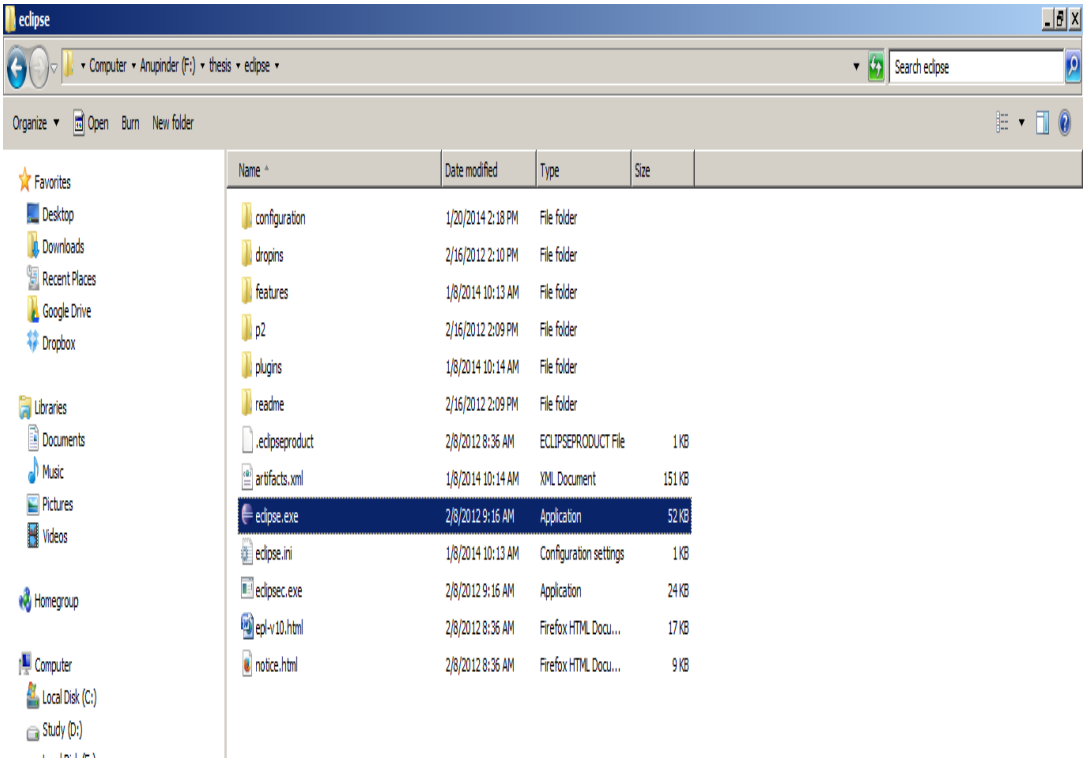
1. **Cloudlet**: set of tasks ready to submit for processing.
2. **VM (Virtual Machine)**: this term represents a logical image of machine defined with characteristics just like physical machine.
3. **Host**: This will represent a physical machine with specific characteristics. Each host will consist of minimum one or more processing element.
4. **Datacenter**: This will represent a physical setup, which will have more than one interconnected host setup using a specific topology (to form a compute cluster).
5. **Broker**: This will represent a user or machine though which infrastructure (datacenter resources) will be accessed by virtual machines and cloudlets will be allocated to virtual machines for execution.

4. Installing CloudSim with Eclipse

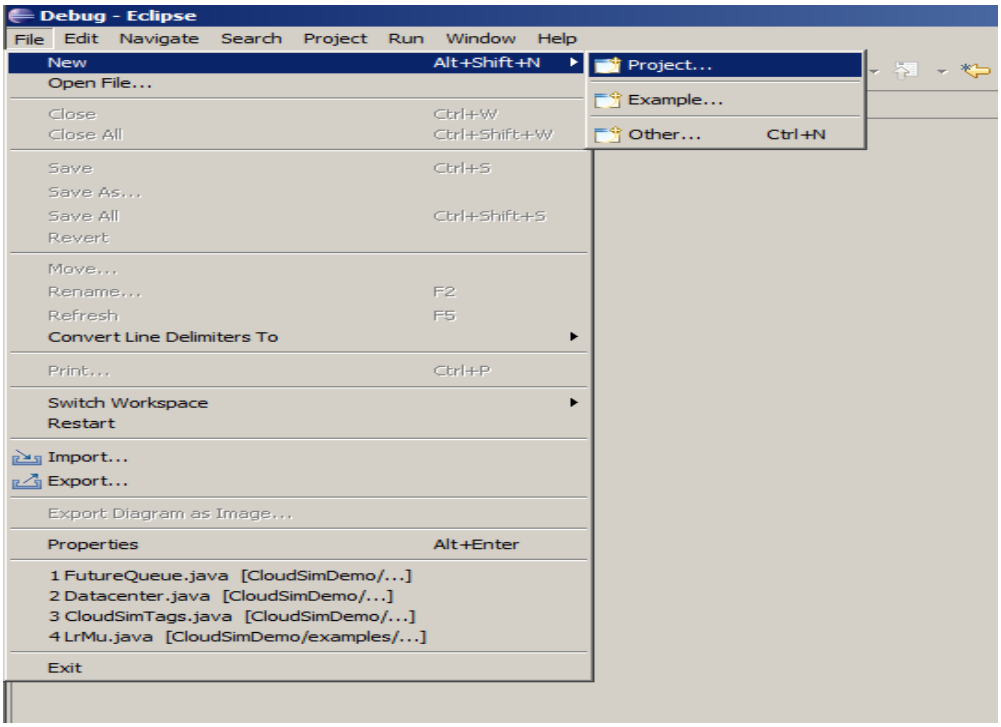
1. Before installing CloudSim following resources must be downloaded on the local system
   a. **Eclipse IDE for java developers**:
      i. Windows 32-bit: http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/juno/SR1/eclipse-java-juno-SR1-win32.zip
      ii. Windows 64-bit: http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/juno/SR1/eclipse-java-juno-SR1-win32-x86_64.zip
   b. **CloudSim Project code**: CloudSim code can be downloaded from following project page https://code.google.com/p/cloudsim/, always download latest version to avoid any bug issues of previous packages.
   c. **One external requirement of cloudsim** i.e. common jar package of math related functions is to be downloaded from http://www.trieuvan.com/apache//commons/math/binaries/commons-math3-3.2-bin.zip

2. Unzip **eclipse** and **cloudsim** to some common folder.

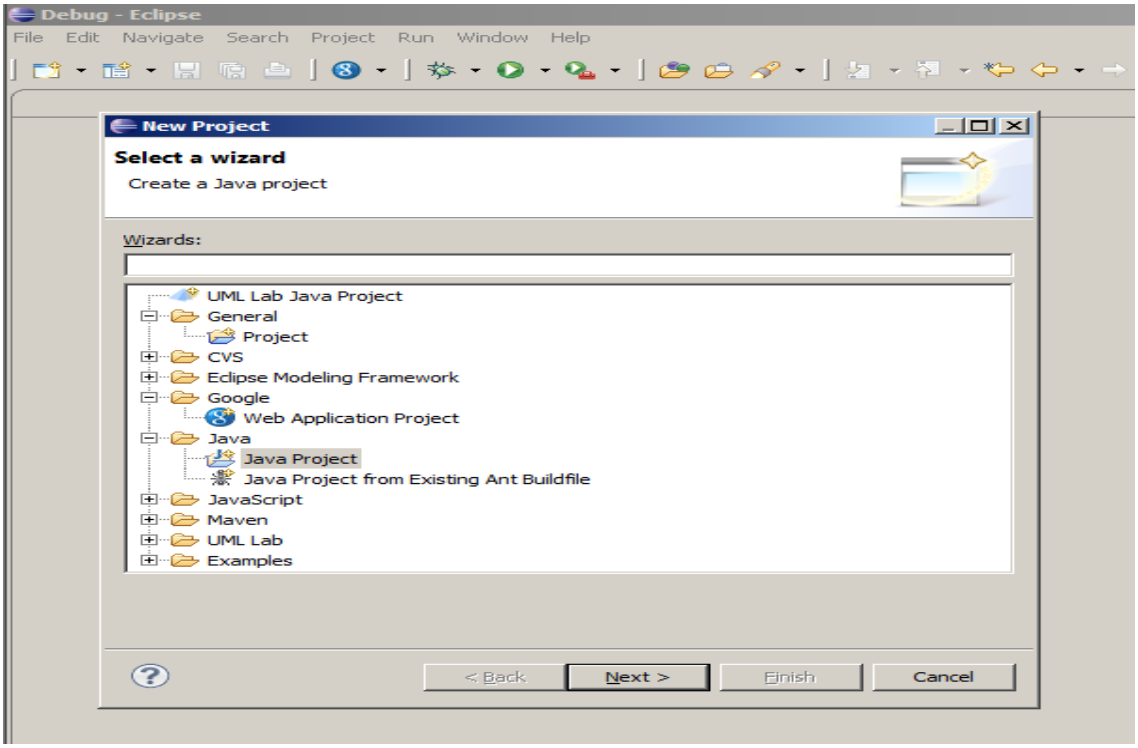
3. Open **eclipse.exe** from eclipse folder:



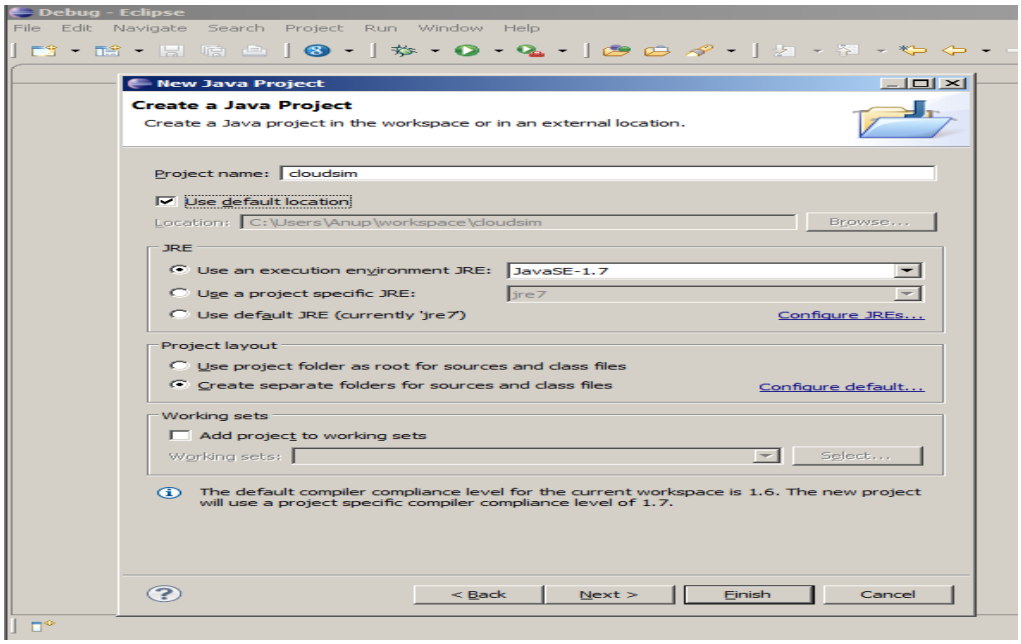
4. Now in Eclipse go to menu File-> New -> Project.

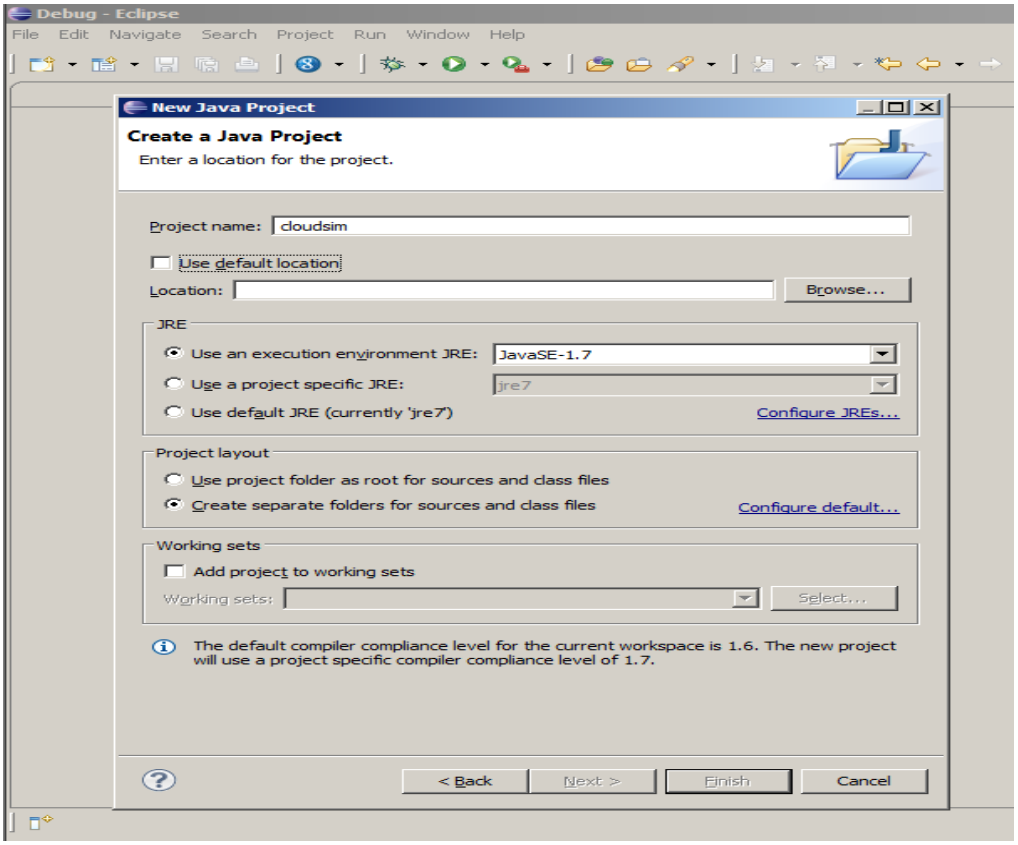


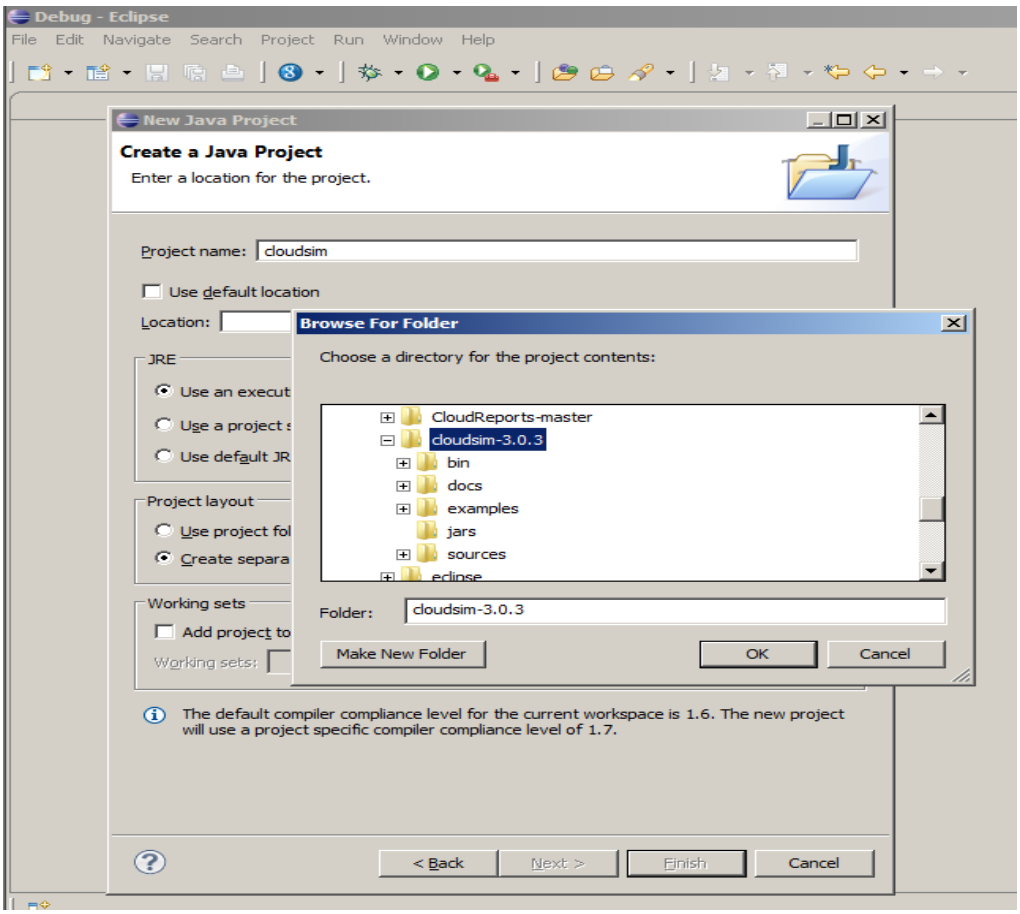
5. Now a project wizard will be opened. From this wizard choose 'Java Project' option and click next.

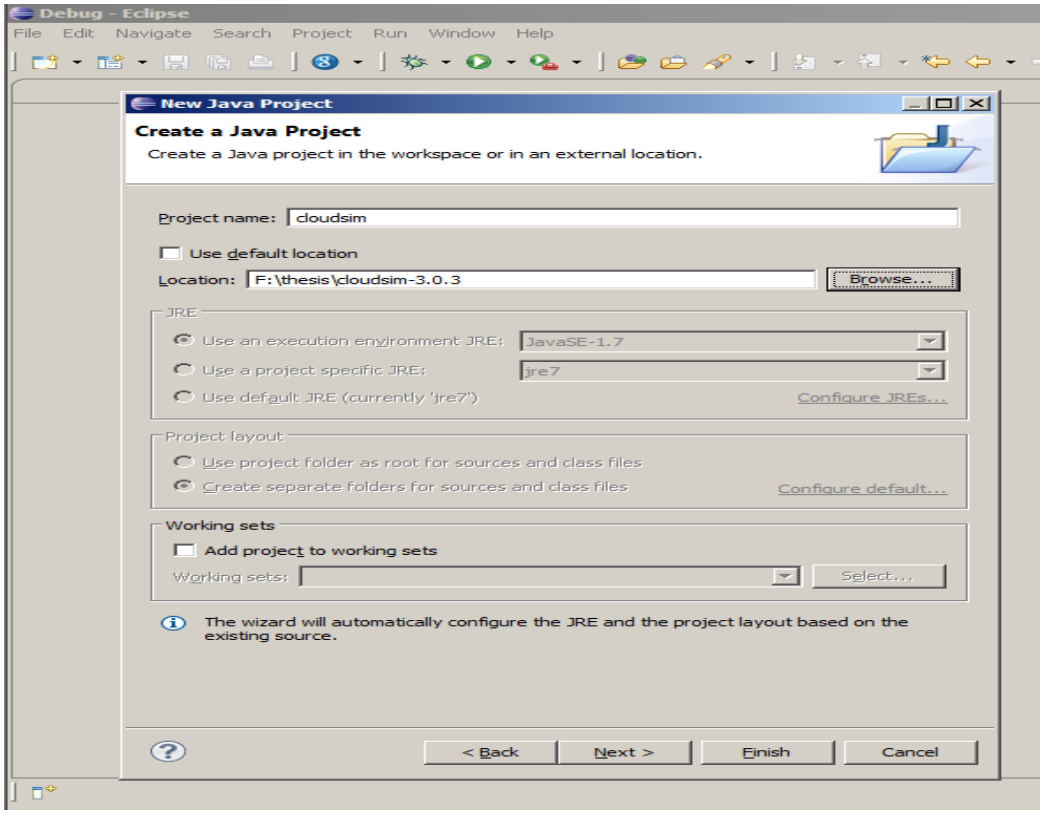


6. Now a detailed new project window will be opened, here you provide project name and the path where you have unzipped the CloudSim project source code or you can put following details as specified:
   a. **Project Name**: CloudSim.
   b. Unselect the 'Use default location' option and then click on browse to open the path where you have unzipped the cloudsim project and finally click Next to set project settings

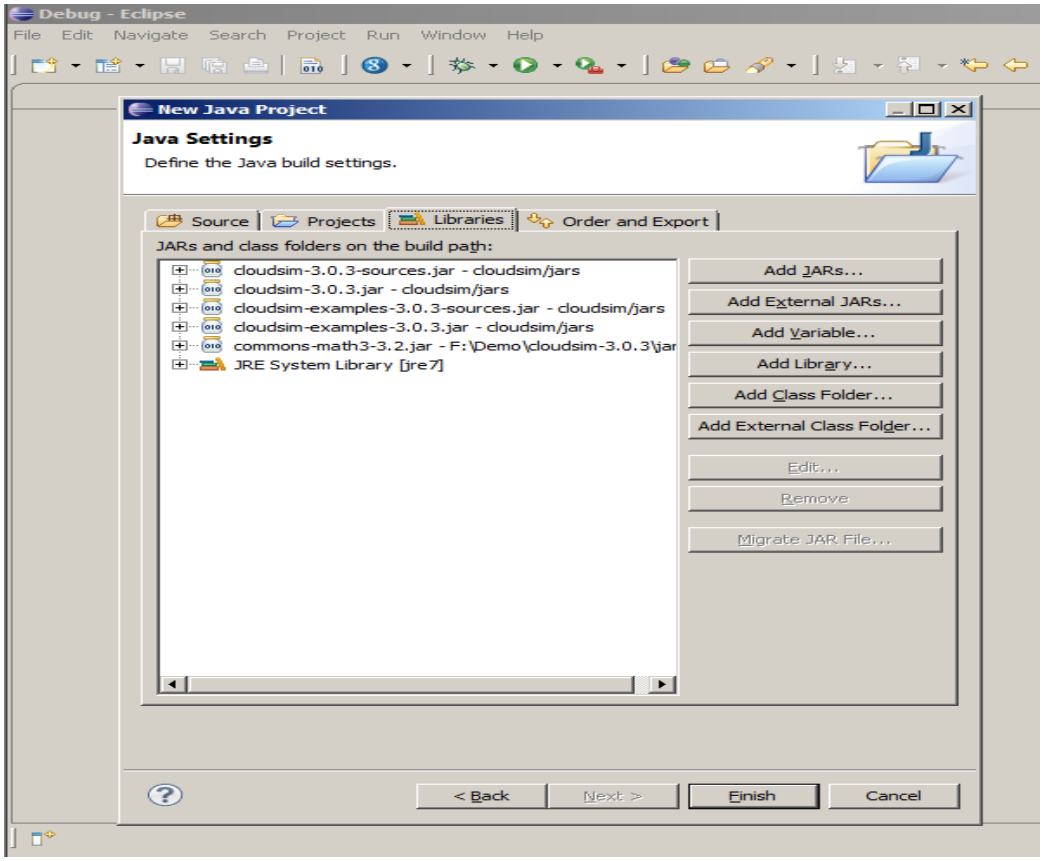



Debug - Eclipse
File Edit Navigate Search Project Run Window Help
New Java Project
Create a Java Project
Enter a location for the project.
Project name: cloudsim
Use default location
Location:
Browse...
JRE
Use an execution environment JRE: JavaSE-1.7
Use a project specific JRE: jre7
Use default JRE (currently 'jre7')
Configure JREs...
Project layout
Use project folder as root for sources and class files
Create separate folders for sources and class files
Configure default...
Working sets
Add project to working sets
Working sets:
Select...
The default compiler compliance level for the current workspace is 1.6. The new project will use a project specific compiler compliance level of 1.7.
< Back
Next >
Finish
Cancel

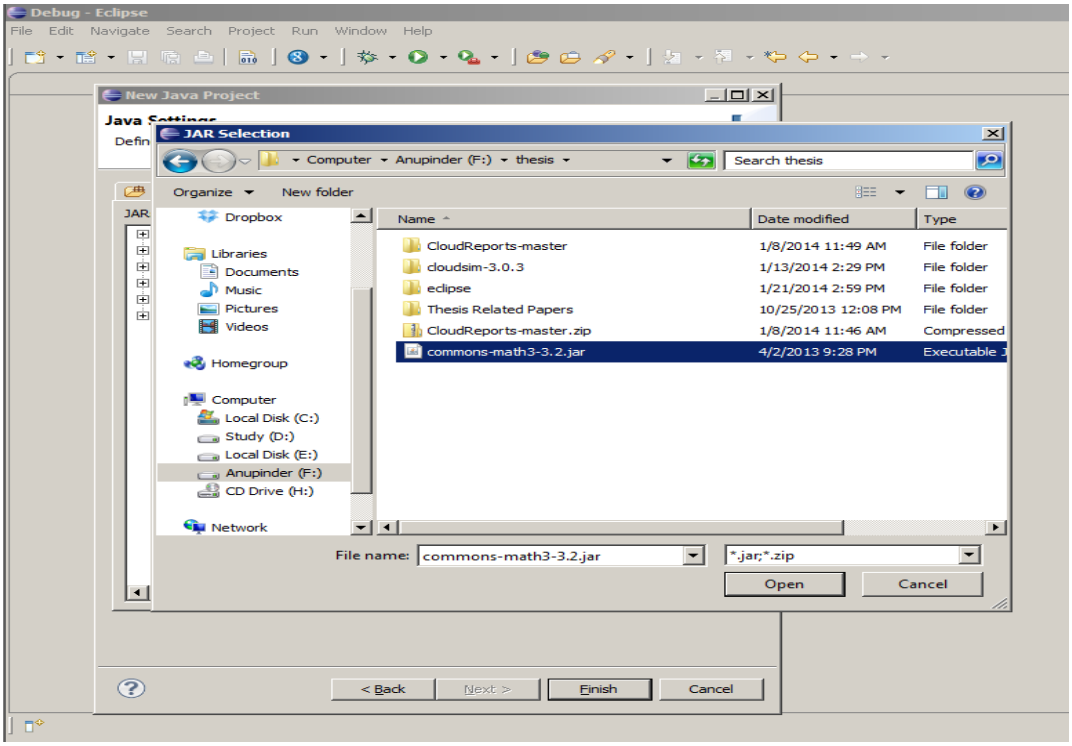

Debug - Eclipse
File Edit Navigate Search Project Run Window Help
New Java Project
Create a Java Project
Enter a location for the project.
Project name: cloudsim
Use default location
Location:
Browse For Folder
Choose a directory for the project contents:
CloudReports-master
cloudsim-3.0.3
bin
docs
examples
jars
sources
eclipse
Folder: cloudsim-3.0.3
Make New Folder
OK
Cancel
The default compiler compliance level for the current workspace is 1.6. The new project will use a project specific compiler compliance level of 1.7.
< Back
Next >
Finish
Cancel



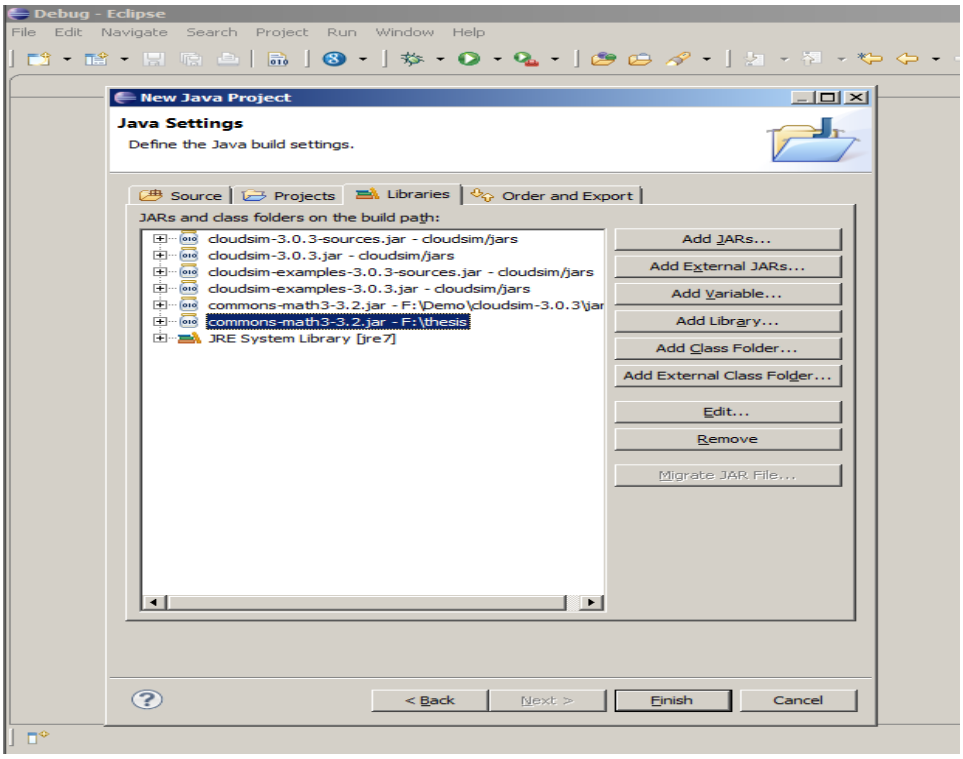
c. Now open 'Libraries' tab and click on add external jar (Math common jar will be included in project from this step)



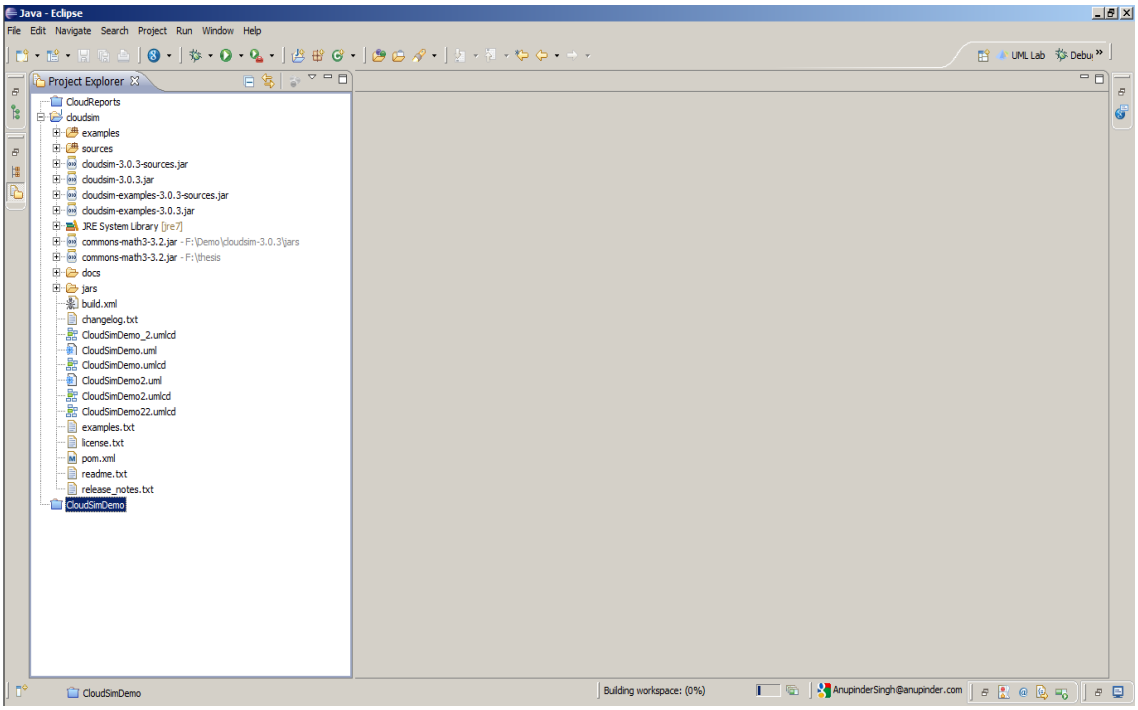
d. Open the path where you have unzipped the math common binaries and select 'Common-math 3.3.2.jar' and click on open.



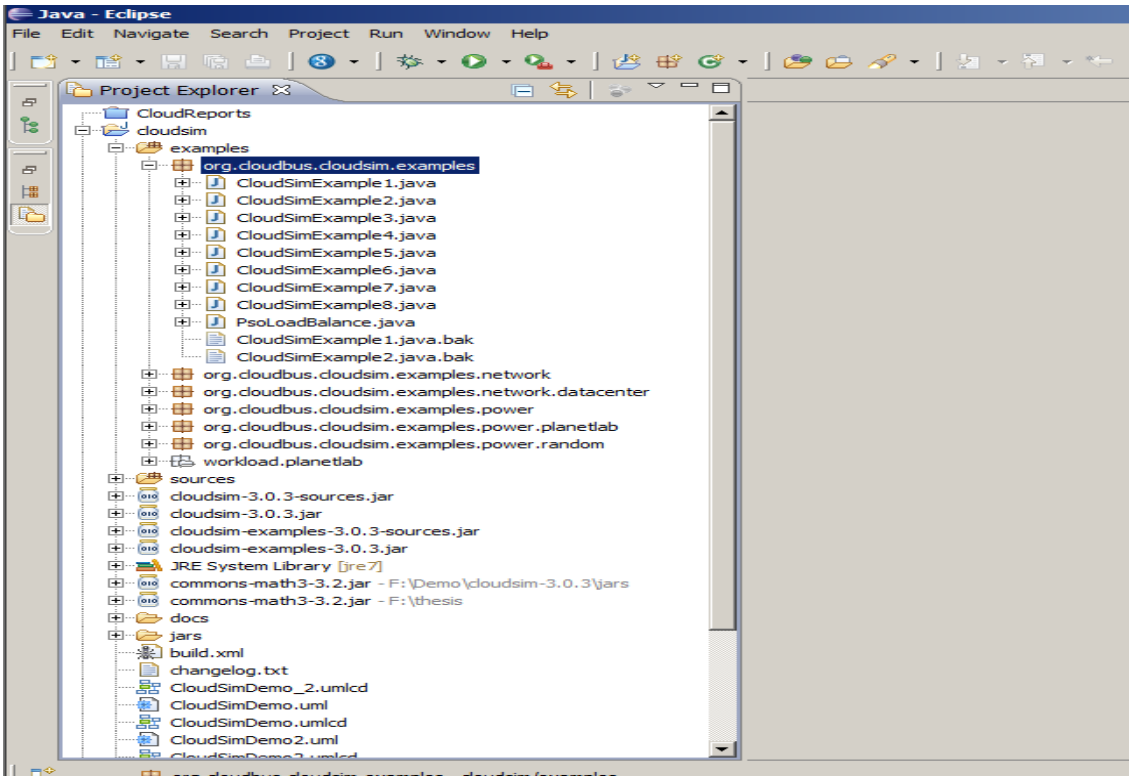
e. Ensure external jar that you opened in previous step is displayed in list and then click on 'Finish' (your system may take 2-3 minutes to configure the project)



7. Once the project is configured you can open the project explorer and start exploring the cloudsim project. Also for the first time eclipse automatically start building the workspace for newly configured cloudsim project, which may take some time depending on the configuration of computer system. Following is the final screen which you will see after cloudsim is configured.



8. In this manual we will be discussing all the examples that are provided under package 'org.cloudbus.cloudsim.examples' (as shown in figure below).



5. How CloudSim Works?

When any example in package 'org.cloudbus.cloudsim.examples' is executed, the sequence of execution will be as follows:

2. CloudSim Example 1

Source code:

```
package org.cloudbus.cloudsim.examples;

/*
 * Title:       CloudSim Toolkit
 * Description:  CloudSim (Cloud Simulation) Toolkit for Modeling and Simulation of Clouds
 * Licence:      GPL - http://www.gnu.org/copyleft/gpl.html
* Copyright (c) 2009, The University of Melbourne, Australia
 */

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.LinkedList;
import java.util.List;

import org.cloudbus.cloudsim.Cloudlet;
import org.cloudbus.cloudsim.CloudletSchedulerTimeShared;
import org.cloudbus.cloudsim.Datacenter;
import org.cloudbus.cloudsim.DatacenterBroker;
import org.cloudbus.cloudsim.DatacenterCharacteristics;
import org.cloudbus.cloudsim.Host;
import org.cloudbus.cloudsim.Log;
import org.cloudbus.cloudsim.Pe;
import org.cloudbus.cloudsim.Storage;
import org.cloudbus.cloudsim.UtilizationModel;
import org.cloudbus.cloudsim.UtilizationModelFull;
import org.cloudbus.cloudsim.Vm;
import org.cloudbus.cloudsim.VmAllocationPolicySimple;
import org.cloudbus.cloudsim.VmSchedulerTimeShared;
import org.cloudbus.cloudsim.core.CloudSim;
import org.cloudbus.cloudsim.provisioners.BwProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.PeProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple;

/**
 * A simple example showing how to create a datacenter with one host and run one
 * cloudlet on it.
 */
public class CloudSimExample1 {

        /** The cloudlet list. */
        private static List<Cloudlet> cloudletList;
        /** The vmlist. */
        private static List<Vm> vmlist;
```

```
/**
 * Creates main() to run this example.
 *
 * @param args
 *          the args
 */

@SuppressWarnings("unused")
public static void main(String[] args) {

        Log.printLine("Starting CloudSimExample1...");
        try {
                // First step: Initialize the CloudSim package. It should be called
                // before creating any entities.
                int num_user = 1; // number of cloud users
                Calendar calendar = Calendar.getInstance();
                boolean trace_flag = false; // mean trace events

                // Initialize the CloudSim library
                CloudSim.init(num_user, calendar, trace_flag);

                // Second step: Create Datacenters
                // Datacenters are the resource providers in CloudSim. We need at
                // list one of them to run a CloudSim simulation
                Datacenter datacenter0 = createDatacenter("Datacenter_0");

                // Third step: Create Broker
                DatacenterBroker broker = createBroker();
                int brokerId = broker.getId();


                // Fourth step: Create one virtual machine
                vmlist = new ArrayList<Vm>();


                // VM description
                int vmid = 0;
                int mips = 1000;
                long size = 10000; // image size (MB)
                int ram = 512; // vm memory (MB)
                long bw = 1000;
                int pesNumber = 1; // number of cpus
                String vmm = "Xen"; // VMM name

                // create VM
                Vm vm = new Vm(vmid, brokerId, mips, pesNumber, ram, bw, size, vmm,
                                new CloudletSchedulerTimeShared());

                // add the VM to the vmList
                vmlist.add(vm);
```

```
                // submit vm list to the broker
                broker.submitVmList(vmlist);

                // Fifth step: Create one Cloudlet
                cloudletList = new ArrayList<Cloudlet>();

                // Cloudlet properties
                int id = 0;
                long length = 400000;
                long fileSize = 300;
                long outputSize = 300;
                UtilizationModel utilizationModel = new UtilizationModelFull();

                Cloudlet cloudlet = new Cloudlet(id, length, pesNumber, fileSize,
                                outputSize, utilizationModel, utilizationModel,
                                utilizationModel);
                cloudlet.setUserId(brokerId);
                cloudlet.setVmId(vmid);

                // add the cloudlet to the list
                cloudletList.add(cloudlet);

                // submit cloudlet list to the broker
                broker.submitCloudletList(cloudletList);

                // Sixth step: Starts the simulation
                CloudSim.startSimulation();

                CloudSim.stopSimulation();

                // Final step: Print results when simulation is over
                List<Cloudlet> newList = broker.getCloudletReceivedList();
                printCloudletList(newList);

                Log.printLine("CloudSimExample1 finished!");
        } catch (Exception e) {
                e.printStackTrace();
                Log.printLine("Unwanted errors happen");
        }
}


/**
 * Creates the datacenter.
 *
 * @param name
 *          the name
 *
 * @return the datacenter
 */

private static Datacenter createDatacenter(String name) {
```

```
// Here are the steps needed to create a PowerDatacenter:
// 1. We need to create a list to store
// our machine
List<Host> hostList = new ArrayList<Host>();

// 2. A Machine contains one or more PEs or CPUs/Cores.
// In this example, it will have only one core.
List<Pe> peList = new ArrayList<Pe>();

int mips = 1000;

// 3. Create PEs and add these into a list.
peList.add(new Pe(0, new PeProvisionerSimple(mips))); // need to store

        // Pe id and

        // MIPS Rating

// 4. Create Host with its id and list of PEs and add them to the list
// of machines
int hostId = 0;
int ram = 2048; // host memory (MB)
long storage = 1000000; // host storage
int bw = 10000;

hostList.add(new Host(hostId, new RamProvisionerSimple(ram),
                    new BwProvisionerSimple(bw), storage, peList,
                    new VmSchedulerTimeShared(peList))); // This is our machine

// 5. Create a DatacenterCharacteristics object that stores the
// properties of a data center: architecture, OS, list of
// Machines, allocation policy: time- or space-shared, time zone
// and its price (G$/Pe time unit).

String arch = "x86"; // system architecture

String os = "Linux"; // operating system

String vmm = "Xen";

double time_zone = 10.0; // time zone this resource located

double cost = 3.0; // the cost of using processing in this resource

double costPerMem = 0.05; // the cost of using memory in this resource

double costPerStorage = 0.001; // the cost of using storage in this
                                                                    // resource
double costPerBw = 0.0; // the cost of using bw in this resource
LinkedList<Storage> storageList = new LinkedList<Storage>(); // we are

                                // not  adding SAN devices by now
```

```
		DatacenterCharacteristics characteristics = new DatacenterCharacteristics(
				arch, os, vmm, hostList, time_zone, cost, costPerMem,
				costPerStorage, costPerBw);

		// 6. Finally, we need to create a PowerDatacenter object.
		Datacenter datacenter = null;
		try {
			datacenter = new Datacenter(name, characteristics,
					new VmAllocationPolicySimple(hostList), storageList, 0);
		} catch (Exception e) {
			e.printStackTrace();
		}

		return datacenter;
}



// We strongly encourage users to develop their own broker policies, to
// submit vms and cloudlets according  to the specific rules of the simulated scenario


/**
 * Creates the broker.
 *
 * @return the datacenter broker
 */


private static DatacenterBroker createBroker() {
		DatacenterBroker broker = null;
		try {
			broker = new DatacenterBroker("Broker");
		} catch (Exception e) {
			e.printStackTrace();
			return null;
		}
		return broker;
}

/**
 * Prints the Cloudlet objects.
 *
 * @param list
 *          list of Cloudlets
 */

private static void printCloudletList(List<Cloudlet> list) {
		int size = list.size();
		Cloudlet cloudlet;

		String indent = "    ";
		Log.printLine();
```

```
                `Log.printLine("========== OUTPUT ==========");
                Log.printLine("Cloudlet ID" + indent + "STATUS" + indent
                                + "Data center ID" + indent + "VM ID" + indent + "Time"
                                + indent + "Start Time" + indent + "Finish Time");

                DecimalFormat dft = new DecimalFormat("###.##");
                for (int i = 0; i < size; i++) {
                        cloudlet = list.get(i);
                        Log.print(indent + cloudlet.getCloudletId() + indent + indent);

                        if (cloudlet.getCloudletStatus() == Cloudlet.SUCCESS) {
                                Log.print("SUCCESS");

                                Log.printLine(indent + indent + cloudlet.getResourceId()
                                                + indent + indent + indent + cloudlet.getVmId()
                                                + indent + indent
                                                + dft.format(cloudlet.getActualCPUTime()) + indent
                                                + indent + dft.format(cloudlet.getExecStartTime())
                                                + indent + indent
                                                + dft.format(cloudlet.getFinishTime()));
                        }
                }
        }
}
```

CloudSim Example 1 demonstrates how to simulate a Data Center with one host and run one cloudlet on it using cloudsim.

Import Section

This example imports various important classes of CloudSim library to simulate such scenario; each of such import enables to simulate a particular type of behavior. Following are the various imports that are used in this demonstration:

```
import org.cloudbus.cloudsim.Cloudlet;
import org.cloudbus.cloudsim.CloudletSchedulerTimeShared;
import org.cloudbus.cloudsim.Datacenter;
import org.cloudbus.cloudsim.DatacenterBroker;
import org.cloudbus.cloudsim.DatacenterCharacteristics;
import org.cloudbus.cloudsim.Host;
import org.cloudbus.cloudsim.Log;
import org.cloudbus.cloudsim.Pe;
import org.cloudbus.cloudsim.Storage;
import org.cloudbus.cloudsim.UtilizationModel;
import org.cloudbus.cloudsim.UtilizationModelFull;
import org.cloudbus.cloudsim.Vm;
import org.cloudbus.cloudsim.VmAllocationPolicySimple;
import org.cloudbus.cloudsim.VmSchedulerTimeShared;
import org.cloudbus.cloudsim.core.CloudSim;
import org.cloudbus.cloudsim.provisioners.BwProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.PeProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple;
```

Class Components

Variables

Cloudletlist: Declared with a private access modifier, Cloudletlist is List data structure of type "Cloudlet" class (i.e.) it will store objects of Cloudlet class

```
"private static List<Cloudlet> cloudletList;"
```

Vmlist: Declared with a private access modifier, Vmlist is a list data structure of type "VM" class. (i.e.) it will store objects of VM class.

```
"private static List<Vm> vmlist;"
```

Class Methods

createDatacenter method: This method creates required instances of host(computer machines) and assign processing element(s) to the host instances. Then a DataCenter class instance is instantiated and assigns host(s) to the datacenter instance. This method takes "Datacenter name" string as a parameter.

Steps	Description	Program Instructions
1	createDatacenter method defined as	private static Datacenter createDatacenter(String name) {

	private and will return instance of Datacenter class(entity)	
2	As each datacenter will have number of server/host machines, so to maintain list of available hosts, we create an arraylist of 'Host' class with name 'hostList'. This list is a dynamic list and any number of new host machines can be added to it.	List<Host> hostList = **new** ArrayList<Host>();
3	- As each host machine will have number of processing elements (CPUs), so to maintain list of available processing elements for each host, we create an arraylist with the name 'peList'. This list is a dynamic list and any number of new host machines can be added to it. - Also each processing element will have some compute capacity, therefore a variable 'mips' is used to store the mips details of processing element. - Then using the PeProvisionerSimple class the associative processing element is added to 'peList'. PeProvisioningSimple class defines the allocation policy of compute capacity (defined as MIPS) to VMs and is used to allocate and de-allocate provisioned compute capacity (defined as MIPS) to Virtual Machines.	List<Pe> peList = **new** ArrayList<Pe>(); **int** mips = 1000; peList.add(**new** Pe(0, **new** PeProvisionerSimple(mips)));

4	- Like processing elements, hosts also possess some compute characteristics: RAM, storage, bandwidth and processing elements. Same are defined using the following code.
	- The host is added to 'hostlist' by passing characteristics in following sequence:
	1. **hostId** - unique id of host
	2. **RamProvisionerSimple** – this parameter sets the best effort allocation policy for provisioning of RAM to virtual machines
	3. **BwProvisionerSimple** this parameter sets the best effort allocation policy for provisioning of bandwidth to virtual machines.
	4. **storage** – sets the maximum secondary storage for host
	5. **peList** – assigns the list of
		int hostId = 0;
		int ram = 2048; // host memory (MB)
		long storage = 1000000; // host storage
		int bw = 10000;
		
		hostList.add(**new** Host(hostId, **new** RamProvisionerSimple(ram),
		new BwProvisionerSimple(bw), storage, peList,
		new VmSchedulerTimeShared(peList))); // This is our machine

	processing elements to host. 6. **vmSchedulerTimeShared** this defines the allocation policy for VM. **Note: vmScheduler is discussed at end of this document.**	
5	After host machines are created they are allocated to datacenter with their unique characteristics which are follows: 1. **arch** - defines architecture of a hosts 2. **os** - defines operating system running on hosts 3. **vmm** – defines the virtual machine monitor used for Virtual machines 4. **hostList** – sets the list of hosts in a datacenter 5. **time_zone** – defines local time zone of a user that owns this reservation. Time zone should be of range [GMT-12 ... GMT+13] 6. **cost** - defines the cost per sec to use this datacenter 7. **costPerMem -** defines the cost to use memory in this datacenter 8. **costPerStorage** - defines the cost to use storage in this datacenter 9. **costPerBw** - defines the cost per unit of bandwidth	String arch = "x86"; String os = "Linux"; String vmm = "Xen"; **double** time_zone = 10.0; **double** cost = 3.0; **double** costPerMem = 0.05; **double** costPerStorage = 0.001; **double** costPerBw = 0.0; LinkedList<Storage> storageList = new LinkedList<Storage>(); // we are not adding SAN devices by now DatacenterCharacteristics characteristics = **new** DatacenterCharacteristics(arch, os, vmm, hostList, time_zone, cost, costPerMem, costPerStorage, costPerBw);

6	Finally instance of DataCenter class is generated by providing following parameters in sequence:
	1. **name** - defines the name to be associated with this Datacenter
	2. **characteristics** - defines an object of DatacenterCharacteristics
	3. **storageList** - defines a LinkedList of storage elements, for data simulation
	4. **vmAllocationPolicy** - defines the Allocation Policy for VM.
	Note: vmAllocationPolicy is discussed at end of this document.
		Datacenter datacenter = **null**;
		try {
		datacenter = **new** Datacenter(name, characteristics, **new** VmAllocationPolicySimple(hostList), storageList, 0);
		} **catch** (Exception e) {
		e.printStackTrace();
		}
		
		return datacenter;
		}

createBroker method: This method creates an instance of DataCenterBroker class. The user is only required to provide the "name of broker" as string input to the method. This is very important instance as this is an interface through which cloudlets (tasks) are sent to datacenter. In case user wants to implement its own broker policies this is the only place where they are required to program their policies.

*"**private static** DatacenterBroker createBroker()"*

printCloudletList method : This method is defined to print the end results of the simulations. This method takes one parameter i.e. list of cloudlets with their status and prints each cloudlet's (tasks) status in a very structured way.

*"**private static void** printCloudletList(List<Cloudlet> list)"*

Main method: This is the first method from where the base execution starts, with preparing of pre-requisites of simulation and then calls two static method from CloudSim class

"CloudSim.*startSimulation*();"

And then

"CloudSim.*stopSimulation*();"

Description of Program

Flow of method call is specified below that will be done by Main method:

Steps	Description	Program Instructions
1	Initialize CloudSim package with number of user specification, time and trace status	**int** num_user = 1; Calendar calendar = Calendar.*getInstance*(); **boolean** trace_flag = **false**; CloudSim.*init*(num_user, calendar, trace_flag);
2	Create a Data Center, this will be created with some specific characteristics of hosts already mentioned in createDatacenter method	Datacenter datacenter0 = *createDatacenter*("Datacenter_0");
3	Create Broker and get its unique ID in the simulation system (act as identifier for broker).	DatacenterBroker broker = *createBroker*(); **int** brokerId = broker.getId();

4	Create required Virtual machines, these will be created with some specific characteristics and are passed to constructor of Vm class. The constructor invocation will be done by passing parameters in following sequence:
	1. **vmid** unique ID of the VM
	2. **userId** ID of the VM's owner(Here **brokerId**)
	3. **mips** the mips
	4. **pesNumber** number of CPUs
	5. **ram** amount of ram
	6. **bw** amount of bandwidth
	7. **size** amount of storage
	8. **vmm** virtual machine monitor
		int vmid = 0;
		int mips = 1000;
		long size = 10000; // image size (MB)
		int ram = 512; // vm memory (MB)
		long bw = 1000;
		int pesNumber = 1; // number of CPUs
		String vmm = "Xen"; // VMM name
		// create VM
		Vm vm = **new** Vm(vmid, brokerId, mips, pesNumber, ram, bw, size, vmm,**new** CloudletSchedulerTimeShared());
		
		vmlist.add(vm); // add the VM to the vmList

	9. **cloudletScheduler** cloudletScheduler policy for cloudlets **Note: More detail about CloudletScheduler policy is discussed separately at end of this document.**	
5	Now submit list of virtual machines to broker, so that virtual machines can be hosted on the hosts of Datacenter(s) as specified by utilization (execution) model.	broker.submitVmList(*vmlist*);

6	Create required cloudlets with some specific characteristics and are passed to constructor of Cloudlet class. The constructor invocation will be done by passing parameters in following sequence:
	1. **id** the unique ID of this Cloudlet.
	2. **length** the length or size (in MI) of this cloudlet to be executed.
	3. **pesNumber** the processing element number.
	4. **fileSize** the file size (in byte) of this cloudlet BEFORE submitting.
	5. **outputSize** the file size (in byte) of this cloudlet AFTER finish executing.
	6. **utilizationModelCpu** the utilization model cpu(Here it is utilizationModel)
	7. **utilizationModelRam** the utilization model ram(Here it is utilizationModel)
	8. **utilizationModelBw** the utilization model bw (Here it is utilizationModel)
	Note: More detail about UtilizationModel is discussed separately at end of this document.
		cloudletList = **new** ArrayList<Cloudlet>();
		
		// Cloudlet properties
		int id = 0;
		long length = 400000;
		long fileSize = 300;
		long outputSize = 300;
		UtilizationModel utilizationModel = **new** UtilizationModelFull();
		Cloudlet cloudlet = **new** Cloudlet(id, length, pesNumber, fileSize,outputSize, utilizationModel, utilizationModel, utilizationModel);
		cloudlet.setUserId(brokerId);
		cloudlet.setVmId(vmid);
		cloudletList.add(cloudlet); // add the cloudlet to the list

7	Now submit list of cloudlets to broker, so that these can be distributed on virtual machines as specified by utilization (execution) model.	broker.submitCloudletList(*cloudletList*);
8	startSimulation() method will start the actual process of simulation by creating and initializing required entities.	CloudSim.startSimulation();
9	stopSimulation() method will free all the entities that we created using startSimulation() method.	CloudSim.stopSimulation();
10	getCloudletReceivedList() method return a multi-parameter list related to final status of cloudlets. This will be further used by printCloudletList() method.	List<Cloudlet> newList = broker.getCloudletReceivedList();
11	Print final status of cloudlets(tasks), that were executed during simulation	printCloudletList(newList); Log.printLine("CloudSimExample1 finished!");

Supportive Content

VmScheduler is an abstract class that defines and implements the policy used by VMM to share processing power among virtual machines running on specified host. The definition of this abstract class is extended to following types of policies:

1. **VmSchedulerTimeShared**: This class implements the VMM allocation policy that allocates one or more processing elements to a single Virtual machine and allows sharing of processing element by multiple virtual machines. This class also considers the overhead of VM migrations in policy definition.

2. **VmSchedulerSpaceShared**: This class implements the VMM allocation policy that allocates one or more processing elements to a single virtual machine, but this policy implementation does not support sharing of processing elements. Under this allocation policy, if any virtual machine requests a processing element and is not available at that time, the allocation fails.

3. **VmSchedulerTimeSharedOverSubscription**: This is an extended implementation of VMSchedulerTimeShared VMM allocation policy, which allows over-subscription of processing elements by virtual machine(s) (i.e.) the scheduler still allows the allocation of VMs that require more CPU capacity that is available. And this oversubscription results in performance degradation.

VmAllocationPolicy is an abstract class that represents the provisioning policy of hosts to virtual machines in a Datacenter. It supports two-stage commit of reservation of hosts: first, we reserve the host and, once committed by the user, it is effectively allocated. **VmAllocationPolicySimple** is an extended version of basic VmAllocationPolicy, this class specifies to choose the host for VM allocations whose processing elements are underutilized.

VmAllocationPolicy vs VmScheduler: VmAllocationPolicy is used at datacenter level where as VmScheduler works at Host level.

CloudletScheduler is a very important class, which declares and implements the basic policy of scheduling performed by a virtual machine. As this is an abstract class, therefore its full definition is provided in three different classes each with its own extended type of policy implementation:

1. **CloudletSchedulerSpaceShared**: This class implements a policy of scheduling for Virtual machine to execute cloudlets in space shared environment (i.e.) only one task will be executed on virtual machine at a time. It means cloudlets share the same queue and requests are processed 1 at a time per computing core. *Space-sharing* is likely to mean sharing memory space (hard disk, RAM, database)

2. **CloudletSchedulerTimeShared**: This class implements a policy of scheduling for Virtual machine to execute cloudlets in time shared environment (i.e.) more than one cloudlet (tasks) will be submitted to virtual machine and each will get its specified share of time. It means several requests (cloudlets) are processed at once but they must share the computing power of that virtual machine, so they will affect each other's processing time. It basically influences the completion time of a cloudlet in CloudSim. ***Time-sharing*** is probably referring to the concept of sharing executing power (such as CPU, logical processor, GPU).

3. **CloudletSchedulerDynamicWorkload**: This implements a special policy of scheduling for virtual machine assuming that there is just one cloudlet which is working as an online service.

UtilizationModel is an interface which defines basic procedures to be implemented in order to provide fine-grained control over resource usage by a cloudlet. Like CloudletScheduler this class contains the abstract methods, therefore its full definition is provided in three different classes each defining its own type model for utilization of resources by cloudlets:

1. **UtilizationModelFull**: This class implements the simplest model, according to which a Cloudlet always utilizes all the available CPU capacity.

2. **UtilizationModelNull**: This class implements another simple model, according to which a Cloudlet always utilizes zero capacity.

3. **UtilizationModelStochastic**: This class implements a model, according to which a Cloudlet generates random CPU utilization every time frame. This resembles real time cloudlet execution environment.

3. CloudSim Example 2

Source code:

```
/*
 * Title:       CloudSim Toolkit
 * Description:  CloudSim (Cloud Simulation) Toolkit for Modeling and Simulation of Clouds
 * Licence:      GPL - http://www.gnu.org/copyleft/gpl.html
* Copyright (c) 2009, The University of Melbourne, Australia
 */

package org.cloudbus.cloudsim.examples;


import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.LinkedList;
import java.util.List;

import org.cloudbus.cloudsim.Cloudlet;
import org.cloudbus.cloudsim.CloudletSchedulerTimeShared;
import org.cloudbus.cloudsim.Datacenter;
import org.cloudbus.cloudsim.DatacenterBroker;
import org.cloudbus.cloudsim.DatacenterCharacteristics;
import org.cloudbus.cloudsim.Host;
import org.cloudbus.cloudsim.Log;
import org.cloudbus.cloudsim.Pe;
import org.cloudbus.cloudsim.Storage;
import org.cloudbus.cloudsim.UtilizationModel;
import org.cloudbus.cloudsim.UtilizationModelFull;
import org.cloudbus.cloudsim.Vm;
import org.cloudbus.cloudsim.VmAllocationPolicySimple;
import org.cloudbus.cloudsim.VmSchedulerTimeShared;
import org.cloudbus.cloudsim.core.CloudSim;
import org.cloudbus.cloudsim.provisioners.BwProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.PeProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple;


/**
 * A simple example showing how to create
 * a datacenter with one host and run two
 * cloudlets on it. The cloudlets run in
 * VMs with the same MIPS requirements.
 * The cloudlets will take the same time to
 * complete the execution.
 */
public class CloudSimExample2 {
```

```
/** The cloudlet list. */
private static List<Cloudlet> cloudletList;

/** The vmlist. */
private static List<Vm> vmlist;

/**
 * Creates main() to run this example
 */
public static void main(String[] args) {

        Log.printLine("Starting CloudSimExample2...");

    try {
        // First step: Initialize the CloudSim package. It should be called
        // before creating any entities.
        int num_user = 1;   // number of cloud users
        Calendar calendar = Calendar.getInstance();
        boolean trace_flag = false;  // mean trace events

        // Initialize the CloudSim library
        CloudSim.init(num_user, calendar, trace_flag);

        // Second step: Create Datacenters
        //Datacenters are the resource providers in CloudSim. We need at list one of them to
run a CloudSim simulation
        Datacenter datacenter0 = createDatacenter("Datacenter_0");

        //Third step: Create Broker
        DatacenterBroker broker = createBroker();
        int brokerId = broker.getId();

        //Fourth step: Create one virtual machine
        vmlist = new ArrayList<Vm>();

        //VM description
        int vmid = 0;
        int mips = 250;
        long size = 10000; //image size (MB)
        int ram = 512; //vm memory (MB)
        long bw = 1000;
        int pesNumber = 1; //number of cpus
        String vmm = "Xen"; //VMM name

        //create two VMs
        Vm vm1 = new Vm(vmid, brokerId, mips, pesNumber, ram, bw, size, vmm, new
CloudletSchedulerTimeShared());

        vmid++;
        Vm vm2 = new Vm(vmid, brokerId, mips, pesNumber, ram, bw, size, vmm, new
CloudletSchedulerTimeShared());
```

```
//add the VMs to the vmList
vmlist.add(vm1);
vmlist.add(vm2);

//submit vm list to the broker
broker.submitVmList(vmlist);


//Fifth step: Create two Cloudlets
cloudletList = new ArrayList<Cloudlet>();

//Cloudlet properties
int id = 0;
pesNumber=1;
long length = 250000;
long fileSize = 300;
long outputSize = 300;
UtilizationModel utilizationModel = new UtilizationModelFull();

Cloudlet cloudlet1 = new Cloudlet(id, length, pesNumber, fileSize, outputSize,
utilizationModel, utilizationModel, utilizationModel);
cloudlet1.setUserId(brokerId);

id++;
Cloudlet cloudlet2 = new Cloudlet(id, length, pesNumber, fileSize, outputSize,
utilizationModel, utilizationModel, utilizationModel);
cloudlet2.setUserId(brokerId);

//add the cloudlets to the list
cloudletList.add(cloudlet1);
cloudletList.add(cloudlet2);

//submit cloudlet list to the broker
broker.submitCloudletList(cloudletList);


//bind the cloudlets to the vms. This way, the broker
// will submit the bound cloudlets only to the specific VM
broker.bindCloudletToVm(cloudlet1.getCloudletId(),vm1.getId());
broker.bindCloudletToVm(cloudlet2.getCloudletId(),vm2.getId());

// Sixth step: Starts the simulation
CloudSim.startSimulation();


// Final step: Print results when simulation is over
List<Cloudlet> newList = broker.getCloudletReceivedList();

CloudSim.stopSimulation();

printCloudletList(newList);

//Print the debt of each user to each datacenter
```

```
                datacenter0.printDebts();

            Log.printLine("CloudSimExample2 finished!");
        }
        catch (Exception e) {
           e.printStackTrace();
           Log.printLine("The simulation has been terminated due to an unexpected error");
        }
    }

        private static Datacenter createDatacenter(String name){

    // Here are the steps needed to create a PowerDatacenter:
    // 1. We need to create a list to store
        //    our machine
        List<Host> hostList = new ArrayList<Host>();

    // 2. A Machine contains one or more PEs or CPUs/Cores.
        // In this example, it will have only one core.
        List<Pe> peList = new ArrayList<Pe>();

        int mips = 1000;

    // 3. Create PEs and add these into a list.
        peList.add(new Pe(0, new PeProvisionerSimple(mips))); // need to store Pe id and MIPS
Rating

    //4. Create Host with its id and list of PEs and add them to the list of machines
    int hostId=0;
    int ram = 2048; //host memory (MB)
    long storage = 1000000; //host storage
    int bw = 10000;

    hostList.add(
                    new Host(
                            hostId,
                            new RamProvisionerSimple(ram),
                            new BwProvisionerSimple(bw),
                            storage,
                            peList,
                            new VmSchedulerTimeShared(peList)
                        )
                ); // This is our machine


    // 5. Create a DatacenterCharacteristics object that stores the
    //    properties of a data center: architecture, OS, list of
    //    Machines, allocation policy: time- or space-shared, time zone
    //    and its price (G$/Pe time unit).
    String arch = "x86";      // system architecture
    String os = "Linux";          // operating system
    String vmm = "Xen";
    double time_zone = 10.0;        // time zone this resource located
```

```
        double cost = 3.0;              // the cost of using processing in this resource
        double costPerMem = 0.05;       // the cost of using memory in this resource
        double costPerStorage = 0.001;  // the cost of using storage in this resource
        double costPerBw = 0.0;         // the cost of using bw in this resource
        LinkedList<Storage> storageList = new LinkedList<Storage>();  //we are not adding SAN devices by now

        DatacenterCharacteristics characteristics = new DatacenterCharacteristics(
                arch, os, vmm, hostList, time_zone, cost, costPerMem, costPerStorage, costPerBw);


        // 6. Finally, we need to create a PowerDatacenter object.
        Datacenter datacenter = null;
        try {
            datacenter = new Datacenter(name, characteristics, new
            VmAllocationPolicySimple(hostList),  storageList, 0);
        } catch (Exception e) {
            e.printStackTrace();
        }

        return datacenter;
    }

    //We strongly encourage users to develop their own broker policies, to submit vms and cloudlets according to the specific rules of the simulated scenario
    private static DatacenterBroker createBroker(){

        DatacenterBroker broker = null;
        try {
                broker = new DatacenterBroker("Broker");
            } catch (Exception e) {
                e.printStackTrace();
                return null;
            }
            return broker;
    }

    /**
     * Prints the Cloudlet objects
     * @param list  list of Cloudlets
     */
    private static void printCloudletList(List<Cloudlet> list) {
        int size = list.size();
        Cloudlet cloudlet;

        String indent = "    ";
        Log.printLine();
        Log.printLine("========== OUTPUT ==========");
        Log.printLine("Cloudlet ID" + indent + "STATUS" + indent +
                "Data center ID" + indent + "VM ID" + indent + "Time" + indent + "Start Time" + indent + "Finish Time");

        DecimalFormat dft = new DecimalFormat("###.##");
```

```
                for (int i = 0; i < size; i++) {
                    cloudlet = list.get(i);
                    Log.print(indent + cloudlet.getCloudletId() + indent + indent);

                    if (cloudlet.getCloudletStatus() == Cloudlet.SUCCESS){
                        Log.print("SUCCESS");

                      Log.printLine( indent + indent + cloudlet.getResourceId() + indent + indent + indent +
cloudlet.getVmId() +
                              indent + indent + dft.format(cloudlet.getActualCPUTime()) + indent + indent +
dft.format(cloudlet.getExecStartTime())+
                      indent + indent + dft.format(cloudlet.getFinishTime()));
                    }
                }

            }
}
```

CloudSim Example 2 demonstrates how to create a datacenter with one host,two virtual machines and run two cloudlets on it. Both virtual machine (vm1,vm2) has same machine configuration.

Import Section

same as in example 1.

Class Components

same as in example 1.

Class Methods

same as in example 1.

Description of Program

Here point 1, 2, 3 and 5 will be same as in previous examples except points 4 and 6 as two virtual machines will be created.

Steps	Description	Program Instructions

4	Create required Virtual machines, these will be created with some specific characteristics and are passed to constructor of Vm class. The constructor invocation will be done by passing parameters in following sequence:
	1. **vmid** unique ID of the VM
	2. **userId** ID of the VM's owner(Here **brokerId**)
	3. **mips** the mips
	4. **pesNumber** number of CPUs
	5. **ram** amount of ram
	6. **bw** amount of bandwidth
	7. **size** amount of storage
	8. **vmm** virtual machine monitor
	9. **cloudletScheduler** cloudletScheduler policy for cloudlets
	Note: More detail about CloudLetScheduler policy is discussed separately at end of this document.
		Here two virtual machine of same configuration will be created.
		
		int vmid = 0;
		int mips = 1000;
		long size = 10000; // image size (MB)
		int ram = 512; // vm memory (MB)
		long bw = 1000;
		int pesNumber = 1; // number of cpus
		String vmm = "Xen"; // VMM name
		/ create VM1
		Vm **vm1** = **new** Vm(vmid, brokerId, mips, pesNumber, ram, bw, size, vmm,**new** CloudletSchedulerTimeShared());
		vmlist.add(vm1); // add the VM to the vmList
		
		// create VM2
		Vm **vm2** = **new** Vm(vmid, brokerId, mips, pesNumber, ram, bw, size, vmm,**new** CloudletSchedulerTimeShared());
		
		vmlist.add(vm2); // add the VM to the vmList

6	Create required cloudlets with some specific characteristics and are passed to constructor of Cloudlet class. The constructor invocation will be done by passing parameters in following sequence:
	1. **id** the unique ID of this Cloudlet.
	2. **length** the length or size (in MI) of this cloudlet to be executed.
		Here two cloudlet will be created as cloudlet1,cloudlet2.
		cloudletList = **new** ArrayList<Cloudlet>();
		
		//Cloudlet properties
		int id = 0;
		pesNumber=1;

	3. **pesNumber** the processing element number.
	4. **fileSize** the file size (in byte) of this cloudlet BEFORE submitting.
	5. **outputSize** the file size (in byte) of this cloudlet AFTER finish executing.
	6. **utilizationModelCpu** the utilization model cpu(Here it is utilizationModel)
	7. **utilizationModelRam** the utilization model ram(Here it is utilizationModel)
	8. **utilizationModelBw** the utilization model bw (Here it is utilizationModel)
	Note: More detail about UtilizationModel is discussed separately at end of this document.
		long length = 250000;
		long fileSize = 300;
		long outputSize = 300;
		UtilizationModel utilizationModel = new UtilizationModelFull();
		
		Cloudlet **cloudlet1** = new Cloudlet(id, length, pesNumber, fileSize, outputSize, utilizationModel, utilizationModel,utilizationModel);
		cloudlet1.setUserId(brokerId);
		id++;
		Cloudlet **cloudlet2** = new Cloudlet(id, length, pesNumber, fileSize, outputSize, utilizationModel, utilizationModel, utilizationModel);
		cloudlet2.setUserId(brokerId);
		//add the cloudlets to the list
		cloudletList.add(cloudlet1);
		cloudletList.add(cloudlet2);
		//submit cloudlet list to the broker
		broker.submitCloudletList(cloudletList);
		//bind the cloudlets to the vms. This way, the broker
		// will submit the bound cloudlets only to the specific VM
		broker.bindCloudletToVm(cloudlet1.getCloudletId(),vm1.getId());
		broker.bindCloudletToVm(cloudlet2.getCloudletId(),vm2.getId());
		// Sixth step: Starts the simulation
		CloudSim.startSimulation();

4. CloudSim Example 3

Source code:

```
/*
 * Title:       CloudSim Toolkit
 * Description:  CloudSim (Cloud Simulation) Toolkit for Modeling and Simulation of Clouds
 * Licence:      GPL - http://www.gnu.org/copyleft/gpl.html
* Copyright (c) 2009, The University of Melbourne, Australia
 */

package org.cloudbus.cloudsim.examples;

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.LinkedList;
import java.util.List;

import org.cloudbus.cloudsim.Cloudlet;
import org.cloudbus.cloudsim.CloudletSchedulerTimeShared;
import org.cloudbus.cloudsim.Datacenter;
import org.cloudbus.cloudsim.DatacenterBroker;
import org.cloudbus.cloudsim.DatacenterCharacteristics;
import org.cloudbus.cloudsim.Host;
import org.cloudbus.cloudsim.Log;
import org.cloudbus.cloudsim.Pe;
import org.cloudbus.cloudsim.Storage;
import org.cloudbus.cloudsim.UtilizationModel;
import org.cloudbus.cloudsim.UtilizationModelFull;
import org.cloudbus.cloudsim.Vm;
import org.cloudbus.cloudsim.VmAllocationPolicySimple;
import org.cloudbus.cloudsim.VmSchedulerTimeShared;
import org.cloudbus.cloudsim.core.CloudSim;
import org.cloudbus.cloudsim.provisioners.BwProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.PeProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple;


/**
 * A simple example showing how to create  a datacenter with two hosts and run two  cloudlets on it.
The cloudlets run in VMs with different MIPS requirements.  The cloudlets will take different time  to
complete the execution depending on the requested VM performance.
 */
public class CloudSimExample3 {

        /** The cloudlet list. */
        private static List<Cloudlet> cloudletList;
```

```
/** The vmlist. */
private static List<Vm> vmlist;

/**
 * Creates main() to run this example
 */
public static void main(String[] args) {

	Log.printLine("Starting CloudSimExample3...");

	try {
		// First step: Initialize the CloudSim package. It should be called
		// before creating any entities.
		int num_user = 1;   // number of cloud users
		Calendar calendar = Calendar.getInstance();
		boolean trace_flag = false;  // mean trace events

		// Initialize the CloudSim library
		CloudSim.init(num_user, calendar, trace_flag);

		// Second step: Create Datacenters
		//Datacenters are the resource providers in CloudSim. We need at list one of them to run a CloudSim simulation
		Datacenter datacenter0 = createDatacenter("Datacenter_0");

		//Third step: Create Broker
		DatacenterBroker broker = createBroker();
		int brokerId = broker.getId();

		//Fourth step: Create one virtual machine
		vmlist = new ArrayList<Vm>();

		//VM description
		int vmid = 0;
		int mips = 250;
		long size = 10000; //image size (MB)
		int ram = 2048; //vm memory (MB)
		long bw = 1000;
		int pesNumber = 1; //number of cpus
		String vmm = "Xen"; //VMM name

		//create two VMs
		Vm vm1 = new Vm(vmid, brokerId, mips, pesNumber, ram, bw, size, vmm, new CloudletSchedulerTimeShared());

		//the second VM will have twice the priority of VM1 and so will receive twice CPU time
		vmid++;
		Vm vm2 = new Vm(vmid, brokerId, 2*mips , pesNumber, ram, bw, size, vmm, new CloudletSchedulerTimeShared());

		//add the VMs to the vmList
```

```
                vmlist.add(vm1);
                vmlist.add(vm2);

                //submit vm list to the broker
                broker.submitVmList(vmlist);


                //Fifth step: Create two Cloudlets
                cloudletList = new ArrayList<Cloudlet>();

                //Cloudlet properties
                int id = 0;
                long length = 40000;
                long fileSize = 300;
                long outputSize = 300;
                UtilizationModel utilizationModel = new UtilizationModelFull();

                Cloudlet cloudlet1 = new Cloudlet(id, length, pesNumber, fileSize, outputSize,
utilizationModel, utilizationModel, utilizationModel);
                cloudlet1.setUserId(brokerId);

                id++;
                Cloudlet cloudlet2 = new Cloudlet(id, length, pesNumber, fileSize, outputSize,
utilizationModel, utilizationModel, utilizationModel);
                cloudlet2.setUserId(brokerId);

                //add the cloudlets to the list
                cloudletList.add(cloudlet1);
                cloudletList.add(cloudlet2);

                //submit cloudlet list to the broker
                broker.submitCloudletList(cloudletList);


                //bind the cloudlets to the vms. This way, the broker
                // will submit the bound cloudlets only to the specific VM
                broker.bindCloudletToVm(cloudlet1.getCloudletId(),vm1.getId());
                broker.bindCloudletToVm(cloudlet2.getCloudletId(),vm2.getId());

                // Sixth step: Starts the simulation
                CloudSim.startSimulation();


                // Final step: Print results when simulation is over
                List<Cloudlet> newList = broker.getCloudletReceivedList();

                CloudSim.stopSimulation();

        printCloudletList(newList);

                //Print the debt of each user to each datacenter
                datacenter0.printDebts();
```

```
                Log.printLine("CloudSimExample3 finished!");
            }
            catch (Exception e) {
                e.printStackTrace();
                Log.printLine("The simulation has been terminated due to an unexpected error");
            }
        }

        private static Datacenter createDatacenter(String name){

            // Here are the steps needed to create a PowerDatacenter:
            // 1. We need to create a list to store
            //    our machine
            List<Host> hostList = new ArrayList<Host>();

            // 2. A Machine contains one or more PEs or CPUs/Cores.
            // In this example, it will have only one core.
            List<Pe> peList = new ArrayList<Pe>();

            int mips = 1000;

            // 3. Create PEs and add these into a list.
            peList.add(new Pe(0, new PeProvisionerSimple(mips))); // need to store Pe id and MIPS Rating

            //4. Create Hosts with its id and list of PEs and add them to the list of machines
            int hostId=0;
            int ram = 2048; //host memory (MB)
            long storage = 1000000; //host storage
            int bw = 10000;

            hostList.add(
                new Host(
                    hostId,
                    new RamProvisionerSimple(ram),
                    new BwProvisionerSimple(bw),
                    storage,
                    peList,
                    new VmSchedulerTimeShared(peList)
                )
            ); // This is our first machine

            //create another machine in the Data center
            List<Pe> peList2 = new ArrayList<Pe>();

            peList2.add(new Pe(0, new PeProvisionerSimple(mips)));

            hostId++;

            hostList.add(
                new Host(
                    hostId,
```

```
                                new RamProvisionerSimple(ram),
                                new BwProvisionerSimple(bw),
                                storage,
                                peList2,
                                new VmSchedulerTimeShared(peList2)
                            )
                    ); // This is our second machine


                    // 5. Create a DatacenterCharacteristics object that stores the
                    //    properties of a data center: architecture, OS, list of
                    //    Machines, allocation policy: time- or space-shared, time zone
                    //    and its price (G$/Pe time unit).
                    String arch = "x86";      // system architecture
                    String os = "Linux";          // operating system
                    String vmm = "Xen";
                    double time_zone = 10.0;         // time zone this resource located
                    double cost = 3.0;              // the cost of using processing in this resource
                    double costPerMem = 0.05;                // the cost of using memory in this resource
                    double costPerStorage = 0.001;  // the cost of using storage in this resource
                    double costPerBw = 0.0;                                  // the cost of using bw in this resource
                    LinkedList<Storage> storageList = new LinkedList<Storage>();     //we are not adding
SAN devices by now

        DatacenterCharacteristics characteristics = new DatacenterCharacteristics(
                arch, os, vmm, hostList, time_zone, cost, costPerMem, costPerStorage, costPerBw);

                    // 6. Finally, we need to create a PowerDatacenter object.
                    Datacenter datacenter = null;
                    try {
                            datacenter      =      new      Datacenter(name,      characteristics,      new
VmAllocationPolicySimple(hostList), storageList, 0);
                    } catch (Exception e) {
                            e.printStackTrace();
                    }

                    return datacenter;
            }

            //We strongly encourage users to develop their own broker policies, to submit vms and
cloudlets according
            //to the specific rules of the simulated scenario
            private static DatacenterBroker createBroker(){

                    DatacenterBroker broker = null;
                    try {
                            broker = new DatacenterBroker("Broker");
                    } catch (Exception e) {
                            e.printStackTrace();
                            return null;
                    }
                    return broker;
```

```
	}

	/**
	 * Prints the Cloudlet objects
	 * @param list  list of Cloudlets
	 */
	private static void printCloudletList(List<Cloudlet> list) {
		int size = list.size();
		Cloudlet cloudlet;

		String indent = "    ";
		Log.printLine();
		Log.printLine("========== OUTPUT ==========");
		Log.printLine("Cloudlet ID" + indent + "STATUS" + indent +
				"Data center ID" + indent + "VM ID" + indent + "Time" + indent + "Start
Time" + indent + "Finish Time");

		DecimalFormat dft = new DecimalFormat("###.##");
		for (int i = 0; i < size; i++) {
			cloudlet = list.get(i);
			Log.print(indent + cloudlet.getCloudletId() + indent + indent);

			if (cloudlet.getCloudletStatus() == Cloudlet.SUCCESS){
				Log.print("SUCCESS");

				Log.printLine( indent + indent + cloudlet.getResourceId() + indent +
indent + indent + cloudlet.getVmId() +
					indent + indent +
dft.format(cloudlet.getActualCPUTime()) + indent + indent + dft.format(cloudlet.getExecStartTime())+
					indent + indent + dft.format(cloudlet.getFinishTime()));
			}
		}

	}
}
```

CloudSim Example 3 demonstrates how to create a datacenter with two hosts, two virtual machines and run two cloudlets on it. The cloudlets run in VMs with different MIPS requirements. The second VM will have twice the priority of virtual machine one (VM1) and so cloudlet will receive twice CPU time to complete the execution.

Note:- *Here it is carefully observed that everything is same as discussed in example 2 except in second VM creation,* ***the computation power (mips) is twice of VM1*** *,two host and two cloudlet will execute as in example 2. See point number 4 and 6 in "Description of Program" section for above mentioned changes.*

Import Section

same as example 1.

Class Components

same as example 1.

Class Methods

same as example 1.

Important Description

Here point 1, 2, 5 and 6 will be same as in previous examples except point 3 and 4 as two hosts will need to create as per program objective.

Steps	Description	Program Instructions

3	- As each host machine will have number of processing elements (CPUs) and to maintain list of available processing elements for each host, we create an arraylist with name 'peList'. This list is a dynamic list and any number of new host machines can be added to it.
	- Also each processing element will have some compute capacity, therefore a variable 'mips' is used to store the mips details of processing element.
	- Then using the PeProvisionerSimple class the associative processing element is added to 'peList'. PeProvisioningSimple class defines the allocation policy of compute capacity(defined as MIPS) to VMs and is used to allocate and de-allocate provisioned compute capacity (defined as MIPS) to Virtual Machines.
		//create processor list for first host machine in the Data center
		
		List<Pe> peList = **new** ArrayList<Pe>();
		int mips = 1000;
		peList.add(**new** Pe(0, **new** PeProvisionerSimple(mips)));
		
		//create processor list for second host machine in the Data center
		List<Pe> peList2 = new ArrayList<Pe>();
		peList2.add(new Pe(0, new PeProvisionerSimple(mips)));

4	- Like processing element, host also possesses some compute	**int** hostId = 0; **int** ram = 2048; // host memory (MB)

	characteristics as: RAM, storage, bandwidth and processing elements. Same are defined using the following code.
	- The host is added to 'hostlist' by passing characteristics in following sequence:
	1. **hostId** - unique id of host
	2. **RamProvisionerSimple** – this parameter sets the best effort allocation policy for provisioning of RAM to virtual machines
	3. **BwProvisionerSimple** this parameter sets the best effort allocation policy for provisioning of bandwidth to virtual machines.
	4. **storage** – sets the maximum secondary storage for host
	5. **peList** – assigns the list of processing elements to host.
	6. **vmSchedulerTimeShared** this defines the allocation policy for VM.
	Note: vmScheduler is discussed at end of this document.
		long storage = 1000000; // host storage
		int bw = 10000;
		
		hostList.add(**new** Host(hostId, **new** RamProvisionerSimple(ram),
		new BwProvisionerSimple(bw), storage, peList,
		new VmSchedulerTimeShared(peList))); // This is our machine
		
		hostId++;
		
		hostList.add(**new** Host(hostId, **new** RamProvisionerSimple(ram),
		new BwProvisionerSimple(bw), storage, peList2,
		new VmSchedulerTimeShared(peList2))); // This is our second machine
		
		Note:- 'hostList.add' will be called two times because we have to create two host.

In this section, we will discuss only point 4 rest is same as example **1 and 2**.

Steps	Description	Program Instructions

4	Create required Virtual machines, these will be created with some specific characteristics and are passed to constructor of Vm class. The constructor invocation will be done by passing parameters in following sequence:
	1. **vmid** unique ID of the VM
	2. **userId** ID of the VM's owner(Here **brokerId**)
	3. **mips** the mips
	4. **pesNumber** number of CPUs
	5. **ram** amount of ram
	6. **bw** amount of bandwidth
	7. **size** amount of storage
	8. **vmm** virtual machine monitor
	9. **cloudletScheduler** cloudletScheduler policy for
		Here in the creation of second virtual machine (vm2), we will set the value of 'mips' parameter as '*2*mips*'.
		int vmid = 0;
		int mips = 1000;
		long size = 10000; // image size (MB)
		int ram = 512; // vm memory (MB)
		long bw = 1000;
		int pesNumber = 1; // number of cpus
		String vmm = "Xen"; // VMM name
		
		// create VM
		Vm **vm1** = **new** Vm(vmid, brokerId, mips, pesNumber, ram, bw, size, vmm,**new** CloudletSchedulerTimeShared());
		
		vmlist.add(vm1); // add the VM to the vmList
		
		// create VM

	cloudlets **Note: More detail about CloudLetScheduler policy is discussed separately at end of this document.**	Vm **vm2** = **new** Vm(vmid, brokerId, **2*mips**, pesNumber, ram, bw, size, vmm,**new** CloudletSchedulerTimeShared()); *vmlist*.add(vm2); // add the VM to the vmList

5. CloudSim Example 4

Source code:

```
/*
 * Title:       CloudSim Toolkit
 * Description:  CloudSim (Cloud Simulation) Toolkit for Modeling and Simulation of Clouds
 * Licence:      GPL - http://www.gnu.org/copyleft/gpl.html
* Copyright (c) 2009, The University of Melbourne, Australia
 */

package org.cloudbus.cloudsim.examples;

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.LinkedList;
import java.util.List;

import org.cloudbus.cloudsim.Cloudlet;
import org.cloudbus.cloudsim.CloudletSchedulerTimeShared;
import org.cloudbus.cloudsim.Datacenter;
import org.cloudbus.cloudsim.DatacenterBroker;
import org.cloudbus.cloudsim.DatacenterCharacteristics;
import org.cloudbus.cloudsim.Host;
import org.cloudbus.cloudsim.Log;
import org.cloudbus.cloudsim.Pe;
import org.cloudbus.cloudsim.Storage;
import org.cloudbus.cloudsim.UtilizationModel;
import org.cloudbus.cloudsim.UtilizationModelFull;
import org.cloudbus.cloudsim.Vm;
import org.cloudbus.cloudsim.VmAllocationPolicySimple;
import org.cloudbus.cloudsim.VmSchedulerSpaceShared;
import org.cloudbus.cloudsim.core.CloudSim;
import org.cloudbus.cloudsim.provisioners.BwProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.PeProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple;


/**
 * A simple example showing how to create
 * two datacenters with one host each and
 * run two cloudlets on them.
 */
public class CloudSimExample4 {

        /** The cloudlet list. */
        private static List<Cloudlet> cloudletList;
```

```
/** The vmlist. */
private static List<Vm> vmlist;

/**
 * Creates main() to run this example
 */
public static void main(String[] args) {

	Log.printLine("Starting CloudSimExample4...");

	try {
		// First step: Initialize the CloudSim package. It should be called
		// before creating any entities.
		int num_user = 1;   // number of cloud users
		Calendar calendar = Calendar.getInstance();
		boolean trace_flag = false;  // mean trace events

		// Initialize the GridSim library
		CloudSim.init(num_user, calendar, trace_flag);

		// Second step: Create Datacenters
		//Datacenters are the resource providers in CloudSim. We need at list one of them to run a CloudSim simulation
		Datacenter datacenter0 = createDatacenter("Datacenter_0");
		Datacenter datacenter1 = createDatacenter("Datacenter_1");

		//Third step: Create Broker
		DatacenterBroker broker = createBroker();
		int brokerId = broker.getId();

		//Fourth step: Create one virtual machine
		vmlist = new ArrayList<Vm>();

		//VM description
		int vmid = 0;
		int mips = 250;
		long size = 10000; //image size (MB)
		int ram = 512; //vm memory (MB)
		long bw = 1000;
		int pesNumber = 1; //number of cpus
		String vmm = "Xen"; //VMM name

		//create two VMs
		Vm vm1 = new Vm(vmid, brokerId, mips, pesNumber, ram, bw, size, vmm, new CloudletSchedulerTimeShared());

		vmid++;
		Vm vm2 = new Vm(vmid, brokerId, mips, pesNumber, ram, bw, size, vmm, new CloudletSchedulerTimeShared());

		//add the VMs to the vmList
		vmlist.add(vm1);
```

```
                vmlist.add(vm2);

                //submit vm list to the broker
                broker.submitVmList(vmlist);


                //Fifth step: Create two Cloudlets
                cloudletList = new ArrayList<Cloudlet>();

                //Cloudlet properties
                int id = 0;
                long length = 40000;
                long fileSize = 300;
                long outputSize = 300;
                UtilizationModel utilizationModel = new UtilizationModelFull();

                Cloudlet cloudlet1 = new Cloudlet(id, length, pesNumber, fileSize, outputSize,
utilizationModel, utilizationModel, utilizationModel);
                cloudlet1.setUserId(brokerId);

                id++;
                Cloudlet cloudlet2 = new Cloudlet(id, length, pesNumber, fileSize, outputSize,
utilizationModel, utilizationModel, utilizationModel);
                cloudlet2.setUserId(brokerId);

                //add the cloudlets to the list
                cloudletList.add(cloudlet1);
                cloudletList.add(cloudlet2);

                //submit cloudlet list to the broker
                broker.submitCloudletList(cloudletList);


                //bind the cloudlets to the vms. This way, the broker
                // will submit the bound cloudlets only to the specific VM
                broker.bindCloudletToVm(cloudlet1.getCloudletId(),vm1.getId());
                broker.bindCloudletToVm(cloudlet2.getCloudletId(),vm2.getId());

                // Sixth step: Starts the simulation
                CloudSim.startSimulation();


                // Final step: Print results when simulation is over
                List<Cloudlet> newList = broker.getCloudletReceivedList();

                CloudSim.stopSimulation();

        printCloudletList(newList);

                //Print the debt of each user to each datacenter
                datacenter0.printDebts();
                datacenter1.printDebts();
```

```
                Log.printLine("CloudSimExample4 finished!");
            }
            catch (Exception e) {
                e.printStackTrace();
                Log.printLine("The simulation has been terminated due to an unexpected
error");
            }
        }

        private static Datacenter createDatacenter(String name){

            // Here are the steps needed to create a PowerDatacenter:
            // 1. We need to create a list to store
            //    our machine
            List<Host> hostList = new ArrayList<Host>();

            // 2. A Machine contains one or more PEs or CPUs/Cores.
            // In this example, it will have only one core.
            List<Pe> peList = new ArrayList<Pe>();

            int mips = 1000;

            // 3. Create PEs and add these into a list.
            peList.add(new Pe(0, new PeProvisionerSimple(mips))); // need to store Pe id and MIPS
Rating

            //4. Create Host with its id and list of PEs and add them to the list of machines
            int hostId=0;
            int ram = 2048; //host memory (MB)
            long storage = 1000000; //host storage
            int bw = 10000;


            //in this example, the VMAllocatonPolicy in use is SpaceShared. It means that only one
VM
            //is allowed to run on each Pe. As each Host has only one Pe, only one VM can run on
each Host.
            hostList.add(
                new Host(
                    hostId,
                    new RamProvisionerSimple(ram),
                    new BwProvisionerSimple(bw),
                    storage,
                    peList,
                    new VmSchedulerSpaceShared(peList)
                )
            ); // This is our first machine

            // 5. Create a DatacenterCharacteristics object that stores the
            //    properties of a data center: architecture, OS, list of
            //    Machines, allocation policy: time- or space-shared, time zone
            //    and its price (G$/Pe time unit).
            String arch = "x86";      // system architecture
```

```
		String os = "Linux";		// operating system
		String vmm = "Xen";
		double time_zone = 10.0;		// time zone this resource located
		double cost = 3.0;			// the cost of using processing in this resource
		double costPerMem = 0.05;		// the cost of using memory in this resource
		double costPerStorage = 0.001;	// the cost of using storage in this resource
		double costPerBw = 0.0;			// the cost of using bw in this resource
		LinkedList<Storage> storageList = new LinkedList<Storage>();	//we are not adding SAN devices by now

		DatacenterCharacteristics characteristics = new DatacenterCharacteristics(
				arch, os, vmm, hostList, time_zone, cost, costPerMem, costPerStorage, costPerBw);


		// 6. Finally, we need to create a PowerDatacenter object.
		Datacenter datacenter = null;
		try {
			datacenter = new Datacenter(name, characteristics, new VmAllocationPolicySimple(hostList), storageList, 0);
		} catch (Exception e) {
			e.printStackTrace();
		}

		return datacenter;
	}

	//We strongly encourage users to develop their own broker policies, to submit vms and cloudlets according
	//to the specific rules of the simulated scenario
	private static DatacenterBroker createBroker(){

		DatacenterBroker broker = null;
		try {
			broker = new DatacenterBroker("Broker");
		} catch (Exception e) {
			e.printStackTrace();
			return null;
		}
		return broker;
	}

	/**
	 * Prints the Cloudlet objects
	 * @param list  list of Cloudlets
	 */
	private static void printCloudletList(List<Cloudlet> list) {
		int size = list.size();
		Cloudlet cloudlet;

		String indent = "    ";
		Log.printLine();
		Log.printLine("========== OUTPUT ==========");
		Log.printLine("Cloudlet ID" + indent + "STATUS" + indent +
```

```
                                "Data center ID" + indent + "VM ID" + indent + "Time" + indent + "Start
Time" + indent + "Finish Time");

                DecimalFormat dft = new DecimalFormat("###.##");
                for (int i = 0; i < size; i++) {
                        cloudlet = list.get(i);
                        Log.print(indent + cloudlet.getCloudletId() + indent + indent);

                        if (cloudlet.getCloudletStatus() == Cloudlet.SUCCESS){
                                Log.print("SUCCESS");

                                Log.printLine( indent + indent + cloudlet.getResourceId() + indent +
indent + indent + cloudlet.getVmId() +
                                                indent + indent +
dft.format(cloudlet.getActualCPUTime()) + indent + indent + dft.format(cloudlet.getExecStartTime())+
                                                indent + indent + dft.format(cloudlet.getFinishTime()));
                        }
                }

        }
}
```

CloudSim Example 4 demonstrates how to create ***two datacenters*** with one host each and run two cloudlets on them.

Import Section

same as example 1.

Class Components

same as example 1.

Class Methods

same as example 1.

Important Description

In this section, we will discuss only point **2,** rest is same as example **1 and 2**.

Steps	Description	Program Instructions
2.	Create two Data Centers, these will be created with some specific characteristics of hosts already mentioned in createDatacenter method	Datacenter datacenter0 = createDatacenter("Datacenter_0"); Datacenter datacenter1 = createDatacenter("Datacenter_1");

6. CloudSim Example 5

Source code:

```
package org.cloudbus.cloudsim.examples;
/*
 * Title:       CloudSim Toolkit
 * Description:  CloudSim (Cloud Simulation) Toolkit for Modeling and Simulation of Clouds
 * Licence:      GPL - http://www.gnu.org/copyleft/gpl.html
* Copyright (c) 2009, The University of Melbourne, Australia
 */

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.LinkedList;
import java.util.List;

import org.cloudbus.cloudsim.Cloudlet;
import org.cloudbus.cloudsim.CloudletSchedulerTimeShared;
import org.cloudbus.cloudsim.Datacenter;
import org.cloudbus.cloudsim.DatacenterBroker;
import org.cloudbus.cloudsim.DatacenterCharacteristics;
import org.cloudbus.cloudsim.Host;
import org.cloudbus.cloudsim.Log;
import org.cloudbus.cloudsim.Pe;
import org.cloudbus.cloudsim.Storage;
import org.cloudbus.cloudsim.UtilizationModel;
import org.cloudbus.cloudsim.UtilizationModelFull;
import org.cloudbus.cloudsim.Vm;
import org.cloudbus.cloudsim.VmAllocationPolicySimple;
import org.cloudbus.cloudsim.VmSchedulerSpaceShared;
import org.cloudbus.cloudsim.core.CloudSim;
import org.cloudbus.cloudsim.provisioners.BwProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.PeProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple;


/**
 * A simple example showing how to create two datacenters with one host each and  run cloudlets of two users on them.
 */

public class CloudSimExample5 {

        /** The cloudlet lists. */
        private static List<Cloudlet> cloudletList1;
        private static List<Cloudlet> cloudletList2;
```

```
/** The vmlists. */
private static List<Vm> vmlist1;
private static List<Vm> vmlist2;

/**
 * Creates main() to run this example
 */
public static void main(String[] args) {

        Log.printLine("Starting CloudSimExample5...");

        try {
                // First step: Initialize the CloudSim package. It should be called
                // before creating any entities.
                int num_user = 2;   // number of cloud users
                Calendar calendar = Calendar.getInstance();
                boolean trace_flag = false;  // mean trace events

                // Initialize the CloudSim library
                CloudSim.init(num_user, calendar, trace_flag);

                // Second step: Create Datacenters
                //Datacenters are the resource providers in CloudSim. We need at list one of
                //them to run a CloudSim simulation
                @SuppressWarnings("unused")
                Datacenter datacenter0 = createDatacenter("Datacenter_0");
                @SuppressWarnings("unused")
                Datacenter datacenter1 = createDatacenter("Datacenter_1");

                //Third step: Create Brokers
                DatacenterBroker broker1 = createBroker(1);
                int brokerId1 = broker1.getId();

                DatacenterBroker broker2 = createBroker(2);
                int brokerId2 = broker2.getId();

                //Fourth step: Create one virtual machine for each broker/user
                vmlist1 = new ArrayList<Vm>();
                vmlist2 = new ArrayList<Vm>();

                //VM description
                int vmid = 0;
                int mips = 250;
                long size = 10000; //image size (MB)
                int ram = 512; //vm memory (MB)
                long bw = 1000;
                int pesNumber = 1; //number of cpus
                String vmm = "Xen"; //VMM name

                //create two VMs: the first one belongs to user1
                Vm vm1 = new Vm(vmid, brokerId1, mips, pesNumber, ram, bw, size, vmm, new
                CloudletSchedulerTimeShared());
```

```
//the second VM: this one belongs to user2
Vm vm2 = new Vm(vmid, brokerId2, mips, pesNumber, ram, bw, size, vmm, new
CloudletSchedulerTimeShared());

//add the VMs to the vmlists
vmlist1.add(vm1);
vmlist2.add(vm2);

//submit vm list to the broker
broker1.submitVmList(vmlist1);
broker2.submitVmList(vmlist2);

//Fifth step: Create two Cloudlets
cloudletList1 = new ArrayList<Cloudlet>();
cloudletList2 = new ArrayList<Cloudlet>();

//Cloudlet properties
int id = 0;
long length = 40000;
long fileSize = 300;
long outputSize = 300;
UtilizationModel utilizationModel = new UtilizationModelFull();

Cloudlet cloudlet1 = new Cloudlet(id, length, pesNumber, fileSize, outputSize,
utilizationModel, utilizationModel, utilizationModel);
cloudlet1.setUserId(brokerId1);

Cloudlet cloudlet2 = new Cloudlet(id, length, pesNumber, fileSize, outputSize,
utilizationModel, utilizationModel, utilizationModel);
cloudlet2.setUserId(brokerId2);

//add the cloudlets to the lists: each cloudlet belongs to one user
cloudletList1.add(cloudlet1);
cloudletList2.add(cloudlet2);

//submit cloudlet list to the brokers
broker1.submitCloudletList(cloudletList1);
broker2.submitCloudletList(cloudletList2);

// Sixth step: Starts the simulation
CloudSim.startSimulation();

// Final step: Print results when simulation is over
List<Cloudlet> newList1 = broker1.getCloudletReceivedList();
List<Cloudlet> newList2 = broker2.getCloudletReceivedList();

CloudSim.stopSimulation();
Log.print("=============> User "+brokerId1+"    ");
printCloudletList(newList1);
Log.print("=============> User "+brokerId2+"    ");
printCloudletList(newList2);

Log.printLine("CloudSimExample5 finished!");
```

```
		}
		catch (Exception e) {
			e.printStackTrace();
			Log.printLine("The simulation has been terminated due to an unexpected
			error");
		}
}
private static Datacenter createDatacenter(String name){

		// Here are the steps needed to create a PowerDatacenter:
		// 1. We need to create a list to store our machine
		List<Host> hostList = new ArrayList<Host>();

		// 2. A Machine contains one or more PEs or CPUs/Cores.
		// In this example, it will have only one core.

		List<Pe> peList = new ArrayList<Pe>();

		int mips=1000;

		// 3. Create PEs and add these into a list.
		peList.add(new Pe(0, new PeProvisionerSimple(mips))); // need to store Pe id and MIPS
		Rating

		//4. Create Host with its id and list of PEs and add them to the list of machines
		int hostId=0;
		int ram = 2048; //host memory (MB)
		long storage = 1000000; //host storage
		int bw = 10000;

		//in this example, the VMAllocatonPolicy in use is SpaceShared. It means that only one
		//VM is allowed to run on each Pe. As each Host has only one Pe, only one VM can run
		//on each Host.

		hostList.add( new Host( hostId, new RamProvisionerSimple(ram),
				new BwProvisionerSimple(bw), storage, peList,
				new VmSchedulerSpaceShared(peList))); // This is our first machine

		// 5. Create a DatacenterCharacteristics object that stores the properties of a data
		center: architecture, OS, list of Machines, allocation policy: time- or space-shared,
		time zone and its price (G$/Pe time unit).

		String arch = "x86";	// system architecture
		String os = "Linux";	// operating system
		String vmm = "Xen";
		double time_zone = 10.0;	// time zone this resource located
		double cost = 3.0;		// the cost of using processing in this resource
		double costPerMem = 0.05;		// the cost of using memory in this resource
		double costPerStorage = 0.001;	// the cost of using storage in this resource
		double costPerBw = 0.0;			// the cost of using bw in this resource
		LinkedList<Storage> storageList = new LinkedList<Storage>();	//we are not adding
		SAN devices by now
```

```
        DatacenterCharacteristics characteristics = new DatacenterCharacteristics(
        arch, os, vmm, hostList, time_zone, cost, costPerMem, costPerStorage, costPerBw);


        // 6. Finally, we need to create a PowerDatacenter object.
        Datacenter datacenter = null;
        try {
                datacenter = new Datacenter(name, characteristics, new
                VmAllocationPolicySimple(hostList), storageList, 0);
        } catch (Exception e) {
                e.printStackTrace();
        }

        return datacenter;
}

//We strongly encourage users to develop their own broker policies, to submit vms and
//cloudlets according to the specific rules of the simulated scenario
private static DatacenterBroker createBroker(int id){

        DatacenterBroker broker = null;
        try {
                broker = new DatacenterBroker("Broker"+id);
        } catch (Exception e) {
                e.printStackTrace();
                return null;
        }
        return broker;
}
private static void printCloudletList(List<Cloudlet> list) {
        int size = list.size();
        Cloudlet cloudlet;

        String indent = "    ";
        Log.printLine();
        Log.printLine("========== OUTPUT ==========");
        Log.printLine("Cloudlet ID" + indent + "STATUS" + indent +
                        "Data center ID" + indent + "VM ID" + indent + "Time" + indent + "Start
                        Time" + indent + "Finish Time");

        DecimalFormat dft = new DecimalFormat("###.##");
        for (int i = 0; i < size; i++) {
                cloudlet = list.get(i);
                Log.print(indent + cloudlet.getCloudletId() + indent + indent);

                if (cloudlet.getCloudletStatus() == Cloudlet.SUCCESS){
                        Log.print("SUCCESS");
                        Log.printLine( indent + indent + cloudlet.getResourceId() + indent +
                        indent + indent + cloudlet.getVmId() + indent + indent +
                        dft.format(cloudlet.getActualCPUTime()) + indent + indent +
                        dft.format(cloudlet.getExecStartTime())+indent + indent +
                        dft.format(cloudlet.getFinishTime()));
                }
```

```
                }

            }
        }
```

CloudSim Example 5 demonstrates how to create ***two datacenters*** with one host each and run cloudlets of two users on them.

Import Section

Same as example 1.

Class Components

Same as example 1.

Class Methods

Same as example 1.

Important Description

In this section, the main method is discussed with minor changes in program code to support two user setup.

Steps	Description	Program Instructions

1	Initialize CloudSim package with specification of number of users, time and trace status. There was only one user in previously discussed examples.	`int num_user = 2;`
		`Calendar calendar = Calendar.getInstance();`
		`boolean trace_flag = false;`
		`CloudSim.init(num_user, calendar, trace_flag);`

2	Create two Data Centers, these will be created with some specific characteristics of hosts already mentioned in createDatacenter method.	`Datacenter datacenter0 = createDatacenter("Datacenter_0");` `Datacenter datacenter1 = createDatacenter("Datacenter_1");`

3	Create two Brokers and get there unique ID in the simulation system (act as identifier for broker). These two brokers are created to support two users. Also there is a requirement of two different virtual machine lists to hold user specific allocated virtual machines.	`DatacenterBroker broker1 = createBroker(1);`
		`int brokerId1 = broker1.getId();`
		`DatacenterBroker broker2 = createBroker(2);`
		`int brokerId2 = broker2.getId();`
		`vmlist1 = new ArrayList<Vm>();`
		`vmlist2 = new ArrayList<Vm>();`

4	Create required Virtual machines for each broker, these will be created with some specific characteristics and are passed to constructor of Vm class. The constructor invocation will be done by passing parameters in following sequence:
	1. **vmid** unique ID of the VM
	2. **userId** ID of the VM's owner(Here **brokerId**)
	3. **mips** the mips
	4. **pesNumber** number of CPUs
	5. **ram** amount of ram
	6. **bw** amount of bandwidth
	7. **size** amount of storage
	8. **vmm** virtual machine monitor
	9. **cloudletScheduler** cloudletScheduler policy for cloudlets
	Once VMs are created they will be submitted to broker for further allocation process.
		`int vmid = 0;`
		`int mips = 250;`
		`long size = 10000; // image size (MB)`
		`int ram = 512; // vm memory (MB)`
		`long bw = 1000;`
		`int pesNumber = 1; // number of CPUs`
		`String vmm = "Xen"; // VMM name`
		`// create VM`
		`Vm vm1 = new Vm(vmid, brokerId1, mips, pesNumber, ram, bw, size, vmm,new CloudletSchedulerTimeShared());`
		
		`Vm vm2 = new Vm(vmid, brokerId2, mips, pesNumber, ram, bw, size, vmm,new CloudletSchedulerTimeShared());`
		
		`vmlist1.add(vm1);`
		`vmlist2.add(vm2);// add the VM to the vmList`

	Note: More detail about CloudletScheduler policy is discussed separately at end of this document.	

5	Now submit list of virtual machines to broker, so that virtual machines can be hosted on the hosts of Datacenter(s) as specified by utilization (execution) model.	//submit vm list to the broker
		broker1.submitVmList(vmlist1);
		broker2.submitVmList(vmlist2);

6	Create two lists of cloudlets with some specific characteristics for both users and are passed to constructor of Cloudlet class. The constructor invocation will be done by passing parameters in following sequence:
	9. **id** the unique ID of this Cloudlet.
	10. **length** the length or size (in MI) of this cloudlet to be executed.
	11. **pesNumber** the processing element number.
	12. **fileSize** the file size (in byte) of this cloudlet BEFORE submitting.
	13. **outputSize** the file size (in byte) of this cloudlet AFTER finish executing.
	14. **utilizationModelCpu** the utilization model cpu(Here it is utilizationModel)
	15. **utilizationModelRam** the utilization model ram(Here it is utilizationModel)
	16. **utilizationModelBw** the utilization model bw (Here it is utilizationModel)
	Note: More detail about UtilizationModel is discussed separately at end of this document.
		cloudletList1 = **new** ArrayList<Cloudlet>();
		cloudletList2 = **new** ArrayList<Cloudlet>();
		// Cloudlet properties
		int id = 0;
		long length = 400000;
		long fileSize = 300;
		long outputSize = 300;
		UtilizationModel utilizationModel = **new** UtilizationModelFull();
		
		Cloudlet cloudlet1 = **new** Cloudlet(id, length, pesNumber, fileSize,outputSize, utilizationModel, utilizationModel, utilizationModel);
		cloudlet1.setUserId(brokerId1);
		
		cloudlet cloudlet2 = **new** Cloudlet(id, length, pesNumber, fileSize,outputSize, utilizationModel, utilizationModel, utilizationModel);
		cloudlet2.setUserId(brokerId2);
		
		cloudletList1.add(cloudlet1);
		cloudletList2.add(cloudlet2); // add the cloudlet to the list

7	Now submit list of cloudlets to broker, so that these can be distributed on virtual machines as specified by utilization (execution) model.	broker1.submitCloudletList(*cloudletList1*); broker2.submitCloudletList(*cloudletList2*);
8	startSimulation() method will start the actual process of simulation by creating and initializing required entities.	CloudSim.startSimulation();
9	stopSimulation() method will free all the entities that we created using startSimulation() method.	CloudSim.stopSimulation();
10	getCloudletReceivedList() method returns a multi-parameter list related to final status of cloudlets. This will be further used by printCloudletList() method. As this example runs cloudlets for two users, so two lists will be generated.	List<Cloudlet> newList1 = broker1.getCloudletReceivedList(); List<Cloudlet> newList2 = broker2.getCloudletReceivedList();

11	Print final status of cloudlets(tasks) for both users, those were executed during simulation.	Log.print("==========> User "+brokerId1+" ");
		printCloudletList(newList1);
		
		Log.print("==========> User "+brokerId2+" ");
		printCloudletList(newList2);
		
		Log.printLine("CloudSimExample5 finished!");

7. CloudSim Example 6

Source code:

```
package org.cloudbus.cloudsim.examples;
/*
 * Title:       CloudSim Toolkit
 * Description:  CloudSim (Cloud Simulation) Toolkit for Modeling and Simulation of Clouds
 * Licence:      GPL - http://www.gnu.org/copyleft/gpl.html
* Copyright (c) 2009, The University of Melbourne, Australia
 */

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.LinkedList;
import java.util.List;

import org.cloudbus.cloudsim.Cloudlet;
import org.cloudbus.cloudsim.CloudletSchedulerTimeShared;
import org.cloudbus.cloudsim.Datacenter;
import org.cloudbus.cloudsim.DatacenterBroker;
import org.cloudbus.cloudsim.DatacenterCharacteristics;
import org.cloudbus.cloudsim.Host;
import org.cloudbus.cloudsim.Log;
import org.cloudbus.cloudsim.Pe;
import org.cloudbus.cloudsim.Storage;
import org.cloudbus.cloudsim.UtilizationModel;
import org.cloudbus.cloudsim.UtilizationModelFull;
import org.cloudbus.cloudsim.Vm;
import org.cloudbus.cloudsim.VmAllocationPolicySimple;
import org.cloudbus.cloudsim.VmSchedulerTimeShared;
import org.cloudbus.cloudsim.core.CloudSim;
import org.cloudbus.cloudsim.provisioners.BwProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.PeProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple;

/**
 * An example showing how to create scalable simulations.
 */
public class CloudSimExample6 {
        /** The cloudlet list. */
        private static List<Cloudlet> cloudletList;
        /** The vmlist. */
        private static List<Vm> vmlist;
        private static List<Vm> createVM(int userId, int vms) {
                //Creates a container to store VMs. This list is passed to the broker later
                LinkedList<Vm> list = new LinkedList<Vm>();
                //VM Parameters
```

```
                long size = 10000; //image size (MB)
                int ram = 512; //vm memory (MB)
                int mips = 1000;
                long bw = 1000;
                int pesNumber = 1; //number of cpus
                String vmm = "Xen"; //VMM name

                //create VMs
                Vm[] vm = new Vm[vms];
                for(int i=0;i<vms;i++){
                        vm[i] = new Vm(i, userId, mips, pesNumber, ram, bw, size, vmm, new
                        CloudletSchedulerTimeShared());
                        //for creating a VM with a space shared scheduling policy for cloudlets:
                        //vm[i] = Vm(i, userId, mips, pesNumber, ram, bw, size, priority, vmm, new
                        CloudletSchedulerSpaceShared());
                        list.add(vm[i]);
                }
                return list;
        }

        private static List<Cloudlet> createCloudlet(int userId, int cloudlets){
                // Creates a container to store Cloudlets
                LinkedList<Cloudlet> list = new LinkedList<Cloudlet>();

                //cloudlet parameters
                long length = 1000;
                long fileSize = 300;
                long outputSize = 300;
                int pesNumber = 1;
                UtilizationModel utilizationModel = new UtilizationModelFull();

                Cloudlet[] cloudlet = new Cloudlet[cloudlets];

                for(int i=0;i<cloudlets;i++){
                        cloudlet[i] = new Cloudlet(i, length, pesNumber, fileSize, outputSize,
                        utilizationModel, utilizationModel, utilizationModel);
                        // setting the owner of these Cloudlets
                        cloudlet[i].setUserId(userId);
                        list.add(cloudlet[i]);
                }

                return list;
        }


        ////////////////////////// STATIC METHODS ///////////////////////

        /**
         * Creates main() to run this example
         */
        public static void main(String[] args) {
                Log.printLine("Starting CloudSimExample6...");
```

```
		try {
			// First step: Initialize the CloudSim package. It should be called
			// before creating any entities.
			int num_user = 1;   // number of grid users
			Calendar calendar = Calendar.getInstance();
			boolean trace_flag = false;  // mean trace events

			// Initialize the CloudSim library
			CloudSim.init(num_user, calendar, trace_flag);

			// Second step: Create Datacenters
			//Datacenters are the resource providers in CloudSim. We need at list one of
			them to run a CloudSim simulation
			@SuppressWarnings("unused")
			Datacenter datacenter0 = createDatacenter("Datacenter_0");
			@SuppressWarnings("unused")
			Datacenter datacenter1 = createDatacenter("Datacenter_1");

			//Third step: Create Broker
			DatacenterBroker broker = createBroker();
			int brokerId = broker.getId();

			//Fourth step: Create VMs and Cloudlets and send them to broker
			vmlist = createVM(brokerId,20); //creating 20 vms
			cloudletList = createCloudlet(brokerId,40); // creating 40 cloudlets

			broker.submitVmList(vmlist);
			broker.submitCloudletList(cloudletList);

			// Fifth step: Starts the simulation
			CloudSim.startSimulation();

			// Final step: Print results when simulation is over
			List<Cloudlet> newList = broker.getCloudletReceivedList();

			CloudSim.stopSimulation();

			printCloudletList(newList);

			Log.printLine("CloudSimExample6 finished!");
		}
		catch (Exception e)
		{
			e.printStackTrace();
			Log.printLine("The simulation has been terminated due to an unexpected
			error");
		}
	}

	private static Datacenter createDatacenter(String name){

		// Here are the steps needed to create a PowerDatacenter:
		// 1. We need to create a list to store one or more
```

```
//    Machines
List<Host> hostList = new ArrayList<Host>();

// 2. A Machine contains one or more PEs or CPUs/Cores. Therefore, should
//    create a list to store these PEs before creating
//    a Machine.
List<Pe> peList1 = new ArrayList<Pe>();

int mips = 1000;

// 3. Create PEs and add these into the list.
//for a quad-core machine, a list of 4 PEs is required:
peList1.add(new Pe(0, new PeProvisionerSimple(mips))); // need to store Pe id and MIPS
Rating
peList1.add(new Pe(1, new PeProvisionerSimple(mips)));
peList1.add(new Pe(2, new PeProvisionerSimple(mips)));
peList1.add(new Pe(3, new PeProvisionerSimple(mips)));

//Another list, for a dual-core machine
List<Pe> peList2 = new ArrayList<Pe>();

peList2.add(new Pe(0, new PeProvisionerSimple(mips)));
peList2.add(new Pe(1, new PeProvisionerSimple(mips)));

//4. Create Hosts with its id and list of PEs and add them to the list of machines
int hostId=0;
int ram = 2048; //host memory (MB)
long storage = 1000000; //host storage
int bw = 10000;

hostList.add(
        new Host(
                hostId,
                new RamProvisionerSimple(ram),
                new BwProvisionerSimple(bw),
                storage,
                peList1,
                new VmSchedulerTimeShared(peList1)
        )
); // This is our first machine

hostId++;

hostList.add(
        new Host(
                hostId,
                new RamProvisionerSimple(ram),
                new BwProvisionerSimple(bw),
                storage,
                peList2,
                new VmSchedulerTimeShared(peList2)
        )
); // Second machine
```

```
        //To create a host with a space-shared allocation policy for PEs to VMs:
        //hostList.add(
//              new Host(
//                      hostId,
//                      new CpuProvisionerSimple(peList1),
//                      new RamProvisionerSimple(ram),
//                      new BwProvisionerSimple(bw),
//                      storage,
//                      new VmSchedulerSpaceShared(peList1)
//              )
//      );

        //To create a host with a oportunistic space-shared allocation policy for PEs to VMs:
        //hostList.add(
//              new Host(
//                      hostId,
//                      new CpuProvisionerSimple(peList1),
//                      new RamProvisionerSimple(ram),
//                      new BwProvisionerSimple(bw),
//                      storage,
//                      new VmSchedulerOportunisticSpaceShared(peList1)
//              )
//      );
        // 5. Create a DatacenterCharacteristics object that stores the
        //    properties of a data center: architecture, OS, list of
        //    Machines, allocation policy: time- or space-shared, time zone
        //    and its price (G$/Pe time unit).
        String arch = "x86";     // system architecture
        String os = "Linux";        // operating system
        String vmm = "Xen";
        double time_zone = 10.0;        // time zone this resource located
        double cost = 3.0;              // the cost of using processing in this resource
        double costPerMem = 0.05;                   // the cost of using memory in this resource
        double costPerStorage = 0.1;     // the cost of using storage in this resource
        double costPerBw = 0.1;                                 // the cost of using bw in this resource
        LinkedList<Storage> storageList = new LinkedList<Storage>();     //we are not adding
        SAN devices by now

        DatacenterCharacteristics characteristics = new DatacenterCharacteristics(
         arch, os, vmm, hostList, time_zone, cost, costPerMem, costPerStorage, costPerBw);
        // 6. Finally, we need to create a PowerDatacenter object.
        Datacenter datacenter = null;
        try {
                datacenter      =       new     Datacenter(name,        characteristics,        new
                VmAllocationPolicySimple(hostList), storageList, 0);
        } catch (Exception e) {
                e.printStackTrace();
        }

        return datacenter;
}
```

```
	//We strongly encourage users to develop their own broker policies, to submit vms and cloudlets according
	//to the specific rules of the simulated scenario
	private static DatacenterBroker createBroker(){

		DatacenterBroker broker = null;
		try {
			broker = new DatacenterBroker("Broker");
		} catch (Exception e) {
			e.printStackTrace();
			return null;
		}
		return broker;
	}

	/**
	 * Prints the Cloudlet objects
	 * @param list  list of Cloudlets
	 */
	private static void printCloudletList(List<Cloudlet> list) {
		int size = list.size();
		Cloudlet cloudlet;

		String indent = "    ";
		Log.printLine();
		Log.printLine("========== OUTPUT ==========");
		Log.printLine("Cloudlet ID" + indent + "STATUS" + indent +
				"Data center ID" + indent + "VM ID" + indent + indent + "Time" + indent + "Start Time" + indent + "Finish Time");

		DecimalFormat dft = new DecimalFormat("###.##");
		for (int i = 0; i < size; i++) {
			cloudlet = list.get(i);
			Log.print(indent + cloudlet.getCloudletId() + indent + indent);

			if (cloudlet.getCloudletStatus() == Cloudlet.SUCCESS){
				Log.print("SUCCESS");

				Log.printLine( indent + indent + cloudlet.getResourceId() + indent + indent + indent + cloudlet.getVmId() +
				indent + indent + indent + dft.format(cloudlet.getActualCPUTime()) +
				indent + indent + dft.format(cloudlet.getExecStartTime())+ indent + indent + indent + dft.format(cloudlet.getFinishTime()));
			}
		}

	}
}
```

CloudSim Example 6 demonstrate how to create scalable simulations.

Import Section

Instead of using space share vm allocation policy class

import org.cloudbus.cloudsim.VmSchedulerSpaceShared;

this example uses time share vm allocation policy class

import org.cloudbus.cloudsim.VmSchedulerTimeShared;

Class Components

Same as example 1.

Class Methods

In this example createDataCenter method has been changed to support the scalable simulation environment and two new methods are introduced in this example. These two methods are used to generate a dynamic set of VMs and Cloudlets as required by the configuration of simulation. This is also done to modularize the program and all the code related to VM and Cloudlet configurations from main method is shifted to these two specific methods. Both functions are explained as follows:

createDatacenter method: We have discussed about only those steps in which new set of functionality has been added to this method as compared to example 1 definition and has been highlighted. For more details you can refer to 'Class methods' section of example 1.

Steps	Description	Program Instructions

3	Here, code has been implemented to support behavior of dual/quad core machine. And instead of one list of processing elements, in this example we have two lists.	List<Pe> peList1 = new ArrayList<Pe>();
		int mips = 1000;
		// 3. Create PEs and add these into the list.
		//for a quad-core machine, a list of 4 PEs is required:
		peList1.add(new Pe(0, new PeProvisionerSimple(mips)));
		peList1.add(new Pe(1, new PeProvisionerSimple(mips)));
		peList1.add(new Pe(2, new PeProvisionerSimple(mips)));
		peList1.add(new Pe(3, new PeProvisionerSimple(mips)));
		
		//Another list, for a dual-core machine
		List<Pe> peList2 = new ArrayList<Pe>();
		peList2.add(new Pe(0, new PeProvisionerSimple(mips)));
		peList2.add(new Pe(1, new PeProvisionerSimple(mips)));

4	In comparison to previous examples, one more host is added in the data center with different configuration for processing elements as defined in previous step.	**int** hostId = 0;
		int ram = 2048; // host memory (MB)
		long storage = 1000000; // host storage
		int bw = 10000;
		
		hostList.add(new Host(hostId, new RamProvisionerSimple(ram), new BwProvisionerSimple(bw), storage, peList1,
		new VmSchedulerTimeShared(peList1))); // This is our first machine
		
		hostId++;

		hostList.add(new Host(hostId, new RamProvisionerSimple(ram), new BwProvisionerSimple(bw), storage, peList2, new VmSchedulerTimeShared(peList2))); // Second machine

createVM method: This method creates the container of required instances with specified virtual machines specifications, which is to be hosted on various host machines of datacenter. This takes three inputs: UserId, vms and idShift. UserId is basically the brokerId which we will get from broker instance mentioned in previous method. vms is basically the required number of virtual machines that are required to be instantiated at a moment. Idshift is used to provide a unique seed for sequence number to virtual machines in a broker and default value is 0(zero), but can be any positive value. Also along with all these parameters this method specifies what execution model (time or space shared) will be used.

*"**private static** List<Vm> createVM(**int** userId, **int** vms, **int** idShift)"*

createCloudlet method : This method creates the container of specified instances of cloudlets(tasks) with similar or different specifications. This method specifies what kind of scheduling model (time or space shared) will be used for the tasks. Like previous method, it also takes three parameters: userId, cloudlets and idshift. Here userid and idshift has same role, where as cloudlets parameter will specify maximum number of tasks to create.

*"**private static** List<Cloudlet> createCloudlet(**int** userId, **int** cloudlets, **int** idShift)"*

Important Description

In this example the main method definition is changed and instead of stating configurations of the virtual machines and cloudlets, this method just pass the brokerid and required number of virtual machines or cloudlets to specific function. Changes are highlighted in following program code description.

Steps	Description	Program Instructions
1	Initialize CloudSim package with number of user specification, time and trace status	**int** num_user = 1; Calendar calendar = Calendar.*getInstance*(); **boolean** trace_flag = **false**; CloudSim.*init*(num_user, calendar, trace_flag);
2	Create two Data Centers, these will be created with some specific characteristics of hosts already mentioned in createDatacenter method	@SuppressWarnings("unused") Datacenter datacenter0 = *createDatacenter*("Datacenter_0"); @SuppressWarnings("unused") Datacenter datacenter1 = *createDatacenter*("Datacenter_1");
3	Create Broker and get its unique ID in the simulation system (act as identifier for broker).	DatacenterBroker broker = *createBroker*("Broker_0"); **int** brokerId = broker.getId();
4	Create required Virtual machines; these will be created with some specific characteristics already	*vmlist* = *createVM*(brokerId, 20);

	mentioned in createVM method. In this code 'brokerid' will specify the uniqueid of user,'20' is the number of virtual machines to be created for that broker.	
5	Create required cloudlets these will be created with some specific characteristics already mentioned in createCloudlet method. . In this code 'brokerid' will specify the uniqueid of user,'40' is the number of cloudlets to be created for that broker.	*cloudletList* = *createCloudlet*(brokerId, 40);
6	Now submit list of virtual machines to broker, so that virtual machines can be hosted on the hosts of Datacenter(s) as specified utilization (execution) model.	broker.submitVmList(*vmlist*);
7	Now submit list of cloudlets to broker, so that these can be distributed on virtual machines as specified by utilization (execution) model.	broker.submitCloudletList(*cloudletList*);
8	Now Start the actual simulation process of simulation. This is the event where all the specified entities(data center, host, virtual machine, cloudlets etc are started)	CloudSim.startSimulation();
9	getCloudletReceivedList() method return a multi-parameter list related to final status of cloudlets. This will be further used by printCloudletList() method.	List<Cloudlet> newList = broker.getCloudletReceivedList();
10	Stop Simulation and it will free all the entities that we created using startSimulation() method.	CloudSim.stopSimulation();
11	Print final status of cloudlets(tasks), that were executed during simulation	printCloudletList(newList); Log.printLine("CloudSimExample6 finished!");

8. CloudSim Example 7

Source Code

```
package org.cloudbus.cloudsim.examples;
/*
 * Title:       CloudSim Toolkit
 * Description:  CloudSim (Cloud Simulation) Toolkit for Modeling and Simulation of Clouds
 * Licence:      GPL - http://www.gnu.org/copyleft/gpl.html
* Copyright (c) 2009, The University of Melbourne, Australia
 */

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.LinkedList;
import java.util.List;

import org.cloudbus.cloudsim.Cloudlet;
import org.cloudbus.cloudsim.CloudletSchedulerTimeShared;
import org.cloudbus.cloudsim.Datacenter;
import org.cloudbus.cloudsim.DatacenterBroker;
import org.cloudbus.cloudsim.DatacenterCharacteristics;
import org.cloudbus.cloudsim.Host;
import org.cloudbus.cloudsim.Log;
import org.cloudbus.cloudsim.Pe;
import org.cloudbus.cloudsim.Storage;
import org.cloudbus.cloudsim.UtilizationModel;
import org.cloudbus.cloudsim.UtilizationModelFull;
import org.cloudbus.cloudsim.Vm;
import org.cloudbus.cloudsim.VmAllocationPolicySimple;
import org.cloudbus.cloudsim.VmSchedulerTimeShared;
import org.cloudbus.cloudsim.core.CloudSim;
import org.cloudbus.cloudsim.provisioners.BwProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.PeProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple;

/**
 * An example showing how to pause and resume the simulation,
 * and create simulation entities (a DatacenterBroker in this example)
 * dynamically.
 */
public class CloudSimExample7 {

        /** The cloudlet list. */
        private static List<Cloudlet> cloudletList;

        /** The vmlist. */
        private static List<Vm> vmlist;
```

```
private static List<Vm> createVM(int userId, int vms, int idShift) {
        //Creates a container to store VMs. This list is passed to the broker later
        LinkedList<Vm> list = new LinkedList<Vm>();
        //VM Parameters
        long size = 10000; //image size (MB)
        int ram = 512; //vm memory (MB)
        int mips = 250;
        long bw = 1000;
        int pesNumber = 1; //number of cpus
        String vmm = "Xen"; //VMM name
        //create VMs
        Vm[] vm = new Vm[vms];
        for(int i=0;i<vms;i++)
        {
                vm[i] = new Vm(idShift + i, userId, mips, pesNumber, ram, bw, size, vmm, new
                CloudletSchedulerTimeShared());
                list.add(vm[i]);
        }
        return list;
}
private static List<Cloudlet> createCloudlet(int userId, int cloudlets, int idShift){
        // Creates a container to store Cloudlets
        LinkedList<Cloudlet> list = new LinkedList<Cloudlet>();
        //cloudlet parameters
        long length = 40000;
        long fileSize = 300;
        long outputSize = 300;
        int pesNumber = 1;
        UtilizationModel utilizationModel = new UtilizationModelFull();
        Cloudlet[] cloudlet = new Cloudlet[cloudlets];
        for(int i=0;i<cloudlets;i++)
        {
                cloudlet[i] = new Cloudlet(idShift + i, length, pesNumber, fileSize, outputSize,
                utilizationModel, utilizationModel, utilizationModel);
                // setting the owner of these Cloudlets
                cloudlet[i].setUserId(userId);
                list.add(cloudlet[i]);
        }
        return list;
}
////////////////////////// STATIC METHODS ///////////////////////
/**
 * Creates main() to run this example
 */
public static void main(String[] args) {
        Log.printLine("Starting CloudSimExample7...");
        try {
                // First step: Initialize the CloudSim package. It should be called
                // before creating any entities.
                int num_user = 2;   // number of grid users
                Calendar calendar = Calendar.getInstance();
                boolean trace_flag = false;  // mean trace events
```

```
// Initialize the CloudSim library
CloudSim.init(num_user, calendar, trace_flag);
// Second step: Create Datacenters
//Datacenters are the resource providers in CloudSim. We need at list one of
them to run a CloudSim simulation
@SuppressWarnings("unused")
Datacenter datacenter0 = createDatacenter("Datacenter_0");
@SuppressWarnings("unused")
Datacenter datacenter1 = createDatacenter("Datacenter_1");

//Third step: Create Broker
DatacenterBroker broker = createBroker("Broker_0");
int brokerId = broker.getId();

//Fourth step: Create VMs and Cloudlets and send them to broker
vmlist = createVM(brokerId, 5, 0); //creating 5 vms
cloudletList = createCloudlet(brokerId, 10, 0); // creating 10 cloudlets

broker.submitVmList(vmlist);
broker.submitCloudletList(cloudletList);

// A thread that will create a new broker at 200 clock time
Runnable monitor = new Runnable() {
        @Override
        public void run() {
                CloudSim.pauseSimulation(200);
                while (true) {
                        if (CloudSim.isPaused()) {
                                break;
                        }
                        try {
                                Thread.sleep(100);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }

                Log.printLine("\n\n\n" + CloudSim.clock() + ": The simulation is
                paused for 5 sec \n\n");

                try {
                        Thread.sleep(5000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }

                DatacenterBroker broker = createBroker("Broker_1");
                int brokerId = broker.getId();

                //Create VMs and Cloudlets and send them to broker
                vmlist = createVM(brokerId, 5, 100); //creating 5 vms
                cloudletList = createCloudlet(brokerId, 10, 100); // creating 10
                cloudlets
```

```
                                        broker.submitVmList(vmlist);
                                        broker.submitCloudletList(cloudletList);
                                        CloudSim.resumeSimulation();
                                    }
                                };
                                new Thread(monitor).start();
                                Thread.sleep(1000);
                                // Fifth step: Starts the simulation
                                CloudSim.startSimulation();
                                // Final step: Print results when simulation is over
                                List<Cloudlet> newList = broker.getCloudletReceivedList();
                                CloudSim.stopSimulation();
                                printCloudletList(newList);
                                Log.printLine("CloudSimExample7 finished!");
                        }
                        catch (Exception e)
                        {
                                e.printStackTrace();
                                Log.printLine("The simulation has been terminated due to an unexpected
                                error");
                        }
                }

                private static Datacenter createDatacenter(String name){
                        // Here are the steps needed to create a PowerDatacenter:
                        // 1. We need to create a list to store one or more
                        //    Machines
                        List<Host> hostList = new ArrayList<Host>();
                        // 2. A Machine contains one or more PEs or CPUs/Cores. Therefore, should
                        //    create a list to store these PEs before creating
                        //    a Machine.
                        List<Pe> peList1 = new ArrayList<Pe>();
                        int mips = 1000;
                        // 3. Create PEs and add these into the list.
                        //for a quad-core machine, a list of 4 PEs is required:
                        peList1.add(new Pe(0, new PeProvisionerSimple(mips))); // need to store Pe id and
                        MIPS Rating
                        peList1.add(new Pe(1, new PeProvisionerSimple(mips)));
                        peList1.add(new Pe(2, new PeProvisionerSimple(mips)));
                        peList1.add(new Pe(3, new PeProvisionerSimple(mips)));
                        //Another list, for a dual-core machine
                        List<Pe> peList2 = new ArrayList<Pe>();
                        peList2.add(new Pe(0, new PeProvisionerSimple(mips)));
                        peList2.add(new Pe(1, new PeProvisionerSimple(mips)));
                        //4. Create Hosts with its id and list of PEs and add them to the list of machines
                        int hostId=0;
                        int ram = 16384; //host memory (MB)
                        long storage = 1000000; //host storage
                        int bw = 10000;
                        hostList.add(new Host(hostId, new RamProvisionerSimple(ram),
                                        new BwProvisionerSimple(bw), storage,
                                        peList1, new VmSchedulerTimeShared(peList1))); // our first machine
```

```
            hostId++;
            hostList.add(new Host(hostId, new RamProvisionerSimple(ram),
                                new BwProvisionerSimple(bw),storage, peList2,
                                new VmSchedulerTimeShared(peList2))); // Second machine

            // 5. Create a DatacenterCharacteristics object that stores the
            //    properties of a data center: architecture, OS, list of
            //    Machines, allocation policy: time- or space-shared, time zone
            //    and its price (G$/Pe time unit).
            String arch = "x86";     // system architecture
            String os = "Linux";        // operating system
            String vmm = "Xen";
            double time_zone = 10.0;        // time zone this resource located
            double cost = 3.0;            // the cost of using processing in this resource
            double costPerMem = 0.05;                   // the cost of using memory in this resource
            double costPerStorage = 0.1;     // the cost of using storage in this resource
            double costPerBw = 0.1;                                 // the cost of using bw in this resource
            LinkedList<Storage> storageList = new LinkedList<Storage>();      //we are not adding
            SAN devices by now
            DatacenterCharacteristics characteristics = new DatacenterCharacteristics(
            arch, os, vmm, hostList, time_zone, cost, costPerMem, costPerStorage, costPerBw);
            // 6. Finally, we need to create a PowerDatacenter object.
            Datacenter datacenter = null;
            try {
                    datacenter = new Datacenter(name, characteristics, new
                    VmAllocationPolicySimple(hostList), storageList, 0);
            } catch (Exception e) {
                    e.printStackTrace();
            }
            return datacenter;
    }

    //We strongly encourage users to develop their own broker policies, to submit vms and
    //cloudlets according to the specific rules of the simulated scenario
    private static DatacenterBroker createBroker(String name){

            DatacenterBroker broker = null;
            try {
                    broker = new DatacenterBroker(name);
            } catch (Exception e) {
                    e.printStackTrace();
                    return null;
            }
            return broker;
    }

    /**
     * Prints the Cloudlet objects
     * @param list  list of Cloudlets
     */
    private static void printCloudletList(List<Cloudlet> list) {
            int size = list.size();
            Cloudlet cloudlet;
```

```
                String indent = "    ";
                Log.printLine();
                Log.printLine("========== OUTPUT ==========");
                Log.printLine("Cloudlet ID" + indent + "STATUS" + indent +
                                "Data center ID" + indent + "VM ID" + indent + indent + "Time" + indent
                                + "Start Time" + indent + "Finish Time");

                DecimalFormat dft = new DecimalFormat("###.##");
                for (int i = 0; i < size; i++) {
                        cloudlet = list.get(i);
                        Log.print(indent + cloudlet.getCloudletId() + indent + indent);

                        if (cloudlet.getCloudletStatus() == Cloudlet.SUCCESS){
                                Log.print("SUCCESS");

                                Log.printLine( indent + indent + cloudlet.getResourceId() + indent +
                                indent + indent + cloudlet.getVmId() +
                                indent + indent + indent + dft.format(cloudlet.getActualCPUTime()) +
                                indent + indent + dft.format(cloudlet.getExecStartTime())+ indent +
                                indent + indent + dft.format(cloudlet.getFinishTime()));
                        }
                }

        }
}
```

CloudSim Example 7 demonstrates how the cloudsim simulation can be paused in between as well as how new simulation entities can be created dynamically.

Import Section

Same as Example 6.

Class Components

Same as Example 1.

Class Methods

Same as Example 6.

Important Description

This main method of example 7 is almost similar to example 6. With few minor changes are introduced in code and few additions. Program steps that have been changed are highlighted in following description. Also to avoid confusion with step number we have included the completed code of main method

Steps	Description	Program Instructions

1	Initialize CloudSim package with number of user specification, time and trace status. Instead of one user now we have two users.	`int num_user = 2;`
		`Calendar calendar = Calendar.getInstance();`
		`boolean trace_flag = false;`
		`CloudSim.init(num_user, calendar, trace_flag);`

2	Create two Data Centers, these will be created with some specific characteristics of hosts already mentioned in createDatacenter method	`@SuppressWarnings("unused")`
		`Datacenter datacenter0 = createDatacenter("Datacenter_0");`
		`@SuppressWarnings("unused")`
		`Datacenter datacenter1 = createDatacenter("Datacenter_1");`

3	Create Broker and get its unique ID in the simulation system (act as identifier for broker).	`DatacenterBroker broker = createBroker("Broker_0");` `int brokerId = broker.getId();`
4	Create required Virtual machines, these will be created with some specific characteristics already mentioned in createVM method.Here in addition to brokerid & number of VMs, createVM() method take a seed value from where the virtual machines id will be gererated.	`vmlist = createVM(brokerId, 5, 0);`
5	Create required cloudlets these will be created with some specific characteristics already mentioned in createCloudlet method. Here in addition to	`cloudletList = createCloudlet(brokerId, 10, 0);`

	brokerid & number of cloudlets, createCloudlet() method take a seed value from where the virtual machines id will be gererated.	
6	Now submit list of virtual machines to broker, so that virtual machines can be hosted on the hosts of Datacenter(s) as specified utilization (execution) model.	broker.submitVmList(*vmlist*);
7	Now submit list of cloudlets to broker, so that these can be distributed on virtual machines as specified by utilization (execution) model.	broker.submitCloudletList(*cloudletList*);

8	**This is the new set of code is introduced which** creates a "Runnable" thread instance. This will run in parallel of simulation of existing specifications of virtual machines and cloudlets. Once the startSimulation() is called this set of code lines will pause the simulation after 200 milliseconds for 5 seconds and then a new instance of broker with its new set of virtual machines and cloudlets will be added to current set of instance under execution. Once added the simulation is resumed.	Runnable monitor = new Runnable() {
		@Override
		public void run() {
		CloudSim.pauseSimulation(200);
		while (true) {
		if (CloudSim.isPaused()) {
		break;
		}
		try {
		Thread.sleep(100);
		} catch (InterruptedException e) {
		e.printStackTrace();
		}
		}
		Log.printLine("\n\n\n" + CloudSim.clock() + ": The simulation is paused for 5 sec \n\n");
		
		try {
		Thread.sleep(5000);
		} catch (InterruptedException e) {
		e.printStackTrace();
		}
		
		DatacenterBroker broker = createBroker("Broker_1");
		int brokerId = broker.getId();
		
		vmlist = createVM(brokerId, 5, 100);
		cloudletList = createCloudlet(brokerId, 10, 100);
		broker.submitVmList(vmlist);
		broker.submitCloudletList(cloudletList);
		CloudSim.resumeSimulation();
		}
		};
		new Thread(monitor).start();
		Thread.*sleep*(1000);

9	Now Start the actual simulation process of simulation. This is the event where all the specified entities(data center, host, virtual machine, cloudlets etc	CloudSim.startSimulation();

	are started)	
10	getCloudletReceivedList() method return a multi-parameter list related to final status of cloudlets. This will be further used by printCloudletList() method.	List<Cloudlet> newList = broker.getCloudletReceivedList();
11	Stop Simulation and it will free all the entities that we created using startSimulation() method.	CloudSim.stopSimulation();
12	Print final status of cloudlets(tasks), that were executed during simulation	printCloudletList(newList); Log.printLine("CloudSimExample7 finished!");

9. CloudSim Example 8

Source Code

```
package org.cloudbus.cloudsim.examples;
/*
 * Title:       CloudSim Toolkit
 * Description:  CloudSim (Cloud Simulation) Toolkit for Modeling and Simulation of Clouds
 * Licence:      GPL - http://www.gnu.org/copyleft/gpl.html
* Copyright (c) 2009, The University of Melbourne, Australia
 */

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.LinkedList;
import java.util.List;

import org.cloudbus.cloudsim.Cloudlet;
import org.cloudbus.cloudsim.CloudletSchedulerTimeShared;
import org.cloudbus.cloudsim.Datacenter;
import org.cloudbus.cloudsim.DatacenterBroker;
import org.cloudbus.cloudsim.DatacenterCharacteristics;
import org.cloudbus.cloudsim.Host;
import org.cloudbus.cloudsim.Log;
import org.cloudbus.cloudsim.Pe;
import org.cloudbus.cloudsim.Storage;
import org.cloudbus.cloudsim.UtilizationModel;
import org.cloudbus.cloudsim.UtilizationModelFull;
import org.cloudbus.cloudsim.Vm;
import org.cloudbus.cloudsim.VmAllocationPolicySimple;
import org.cloudbus.cloudsim.VmSchedulerTimeShared;
import org.cloudbus.cloudsim.core.CloudSim;
import org.cloudbus.cloudsim.core.SimEntity;
import org.cloudbus.cloudsim.core.SimEvent;
import org.cloudbus.cloudsim.provisioners.BwProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.PeProvisionerSimple;
import org.cloudbus.cloudsim.provisioners.RamProvisionerSimple;

/**
 * An example showing how to create simulation entities
 * (a DatacenterBroker in this example) in run-time using
 * a globar manager entity (GlobalBroker).
 */
public class CloudSimExample8 {
        /** The cloudlet list. */
        private static List<Cloudlet> cloudletList;
        /** The vmList. */
        private static List<Vm> vmList;
```

```
private static List<Vm> createVM(int userId, int vms, int idShift) {
        //Creates a container to store VMs. This list is passed to the broker later
        LinkedList<Vm> list = new LinkedList<Vm>();
        //VM Parameters
        long size = 10000; //image size (MB)
        int ram = 512; //vm memory (MB)
        int mips = 250;
        long bw = 1000;
        int pesNumber = 1; //number of cpus
        String vmm = "Xen"; //VMM name
        //create VMs
        Vm[] vm = new Vm[vms];
        for(int i=0;i<vms;i++){
                vm[i] = new Vm(idShift + i, userId, mips, pesNumber, ram, bw, size, vmm, new
                CloudletSchedulerTimeShared());
                list.add(vm[i]);
        }

        return list;
}

private static List<Cloudlet> createCloudlet(int userId, int cloudlets, int idShift){
        // Creates a container to store Cloudlets
        LinkedList<Cloudlet> list = new LinkedList<Cloudlet>();

        //cloudlet parameters
        long length = 40000;
        long fileSize = 300;
        long outputSize = 300;
        int pesNumber = 1;
        UtilizationModel utilizationModel = new UtilizationModelFull();

        Cloudlet[] cloudlet = new Cloudlet[cloudlets];

        for(int i=0;i<cloudlets;i++){
                cloudlet[i] = new Cloudlet(idShift + i, length, pesNumber, fileSize, outputSize,
                utilizationModel, utilizationModel, utilizationModel);
                // setting the owner of these Cloudlets
                cloudlet[i].setUserId(userId);
                list.add(cloudlet[i]);
        }
        return list;
}


////////////////////////// STATIC METHODS ///////////////////////

/**
 * Creates main() to run this example
 */
public static void main(String[] args) {
        Log.printLine("Starting CloudSimExample8...");
```

```
		try {
			// First step: Initialize the CloudSim package. It should be called
			// before creating any entities.
			int num_user = 2;   // number of grid users
			Calendar calendar = Calendar.getInstance();
			boolean trace_flag = false;  // mean trace events

			// Initialize the CloudSim library
			CloudSim.init(num_user, calendar, trace_flag);

			GlobalBroker globalBroker = new GlobalBroker("GlobalBroker");

			// Second step: Create Datacenters
			//Datacenters are the resource providers in CloudSim. We need at list one of
			//them to run a CloudSim simulation
			@SuppressWarnings("unused")
			Datacenter datacenter0 = createDatacenter("Datacenter_0");
			@SuppressWarnings("unused")
			Datacenter datacenter1 = createDatacenter("Datacenter_1");

			//Third step: Create Broker
			DatacenterBroker broker = createBroker("Broker_0");
			int brokerId = broker.getId();

			//Fourth step: Create VMs and Cloudlets and send them to broker
			vmList = createVM(brokerId, 5, 0); //creating 5 vms
			cloudletList = createCloudlet(brokerId, 10, 0); // creating 10 cloudlets

			broker.submitVmList(vmList);
			broker.submitCloudletList(cloudletList);

			// Fifth step: Starts the simulation
			CloudSim.startSimulation();

			// Final step: Print results when simulation is over
			List<Cloudlet> newList = broker.getCloudletReceivedList();
			newList.addAll(globalBroker.getBroker().getCloudletReceivedList());

			CloudSim.stopSimulation();

			printCloudletList(newList);

			Log.printLine("CloudSimExample8 finished!");
		}
		catch (Exception e)
		{
			e.printStackTrace();
			Log.printLine("The simulation has been terminated due to an unexpected
			error");
		}
}

private static Datacenter createDatacenter(String name){
```

```
// Here are the steps needed to create a PowerDatacenter:
// 1. We need to create a list to store one or more
//    Machines
List<Host> hostList = new ArrayList<Host>();

// 2. A Machine contains one or more PEs or CPUs/Cores. Therefore, should
//    create a list to store these PEs before creating
//    a Machine.
List<Pe> peList1 = new ArrayList<Pe>();

int mips = 1000;

// 3. Create PEs and add these into the list.
//for a quad-core machine, a list of 4 PEs is required:
peList1.add(new Pe(0, new PeProvisionerSimple(mips))); // need to store Pe id and MIPS
Rating
peList1.add(new Pe(1, new PeProvisionerSimple(mips)));
peList1.add(new Pe(2, new PeProvisionerSimple(mips)));
peList1.add(new Pe(3, new PeProvisionerSimple(mips)));

//Another list, for a dual-core machine
List<Pe> peList2 = new ArrayList<Pe>();

peList2.add(new Pe(0, new PeProvisionerSimple(mips)));
peList2.add(new Pe(1, new PeProvisionerSimple(mips)));

//4. Create Hosts with its id and list of PEs and add them to the list of machines
int hostId=0;
int ram = 16384; //host memory (MB)
long storage = 1000000; //host storage
int bw = 10000;

hostList.add(
        new Host(
                hostId,
                new RamProvisionerSimple(ram),
                new BwProvisionerSimple(bw),
                storage,
                peList1,
                new VmSchedulerTimeShared(peList1)
        )
); // This is our first machine

hostId++;

hostList.add(
        new Host(
                hostId,
                new RamProvisionerSimple(ram),
                new BwProvisionerSimple(bw),
                storage,
                peList2,
```

```
                                new VmSchedulerTimeShared(peList2)
                        )
                ); // Second machine

                // 5. Create a DatacenterCharacteristics object that stores the
                //    properties of a data center: architecture, OS, list of
                //    Machines, allocation policy: time- or space-shared, time zone
                //    and its price (G$/Pe time unit).
                String arch = "x86";     // system architecture
                String os = "Linux";         // operating system
                String vmm = "Xen";
                double time_zone = 10.0;         // time zone this resource located
                double cost = 3.0;              // the cost of using processing in this resource
                double costPerMem = 0.05;                   // the cost of using memory in this resource
                double costPerStorage = 0.1;     // the cost of using storage in this resource
                double costPerBw = 0.1;                                 // the cost of using bw in this resource
                LinkedList<Storage> storageList = new LinkedList<Storage>();      //we are not adding
SAN devices by now

                DatacenterCharacteristics characteristics = new DatacenterCharacteristics(
         arch, os, vmm, hostList, time_zone, cost, costPerMem, costPerStorage, costPerBw);


                // 6. Finally, we need to create a PowerDatacenter object.
                Datacenter datacenter = null;
                try {
                        datacenter = new Datacenter(name, characteristics, new
                        VmAllocationPolicySimple(hostList), storageList, 0);
                } catch (Exception e) {
                        e.printStackTrace();
                }

                return datacenter;
        }

        //We strongly encourage users to develop their own broker policies, to submit vms and
        cloudlets according
        //to the specific rules of the simulated scenario
        private static DatacenterBroker createBroker(String name){

                DatacenterBroker broker = null;
                try {
                        broker = new DatacenterBroker(name);
                } catch (Exception e) {
                        e.printStackTrace();
                        return null;
                }
                return broker;
        }

        /**
         * Prints the Cloudlet objects
         * @param list  list of Cloudlets
```

```
	 */
	private static void printCloudletList(List<Cloudlet> list) {
		int size = list.size();
		Cloudlet cloudlet;

		String indent = "    ";
		Log.printLine();
		Log.printLine("========== OUTPUT ==========");
		Log.printLine("Cloudlet ID" + indent + "STATUS" + indent +
				"Data center ID" + indent + "VM ID" + indent + indent + "Time" + indent
				+ "Start Time" + indent + "Finish Time");

		DecimalFormat dft = new DecimalFormat("###.##");
		for (int i = 0; i < size; i++) {
			cloudlet = list.get(i);
			Log.print(indent + cloudlet.getCloudletId() + indent + indent);

			if (cloudlet.getCloudletStatus() == Cloudlet.SUCCESS){
				Log.print("SUCCESS");

					Log.printLine( indent + indent + cloudlet.getResourceId() +
					indent + indent + indent + cloudlet.getVmId() +
						indent + indent + indent +
					dft.format(cloudlet.getActualCPUTime()) +
						indent + indent +
					dft.format(cloudlet.getExecStartTime())+ indent + indent +
					indent + dft.format(cloudlet.getFinishTime()));
			}
		}

	}

	public static class GlobalBroker extends SimEntity {

		private static final int CREATE_BROKER = 0;
		private List<Vm> vmList;
		private List<Cloudlet> cloudletList;
		private DatacenterBroker broker;

		public GlobalBroker(String name) {
			super(name);
		}

		@Override
		public void processEvent(SimEvent ev) {
			switch (ev.getTag()) {
			case CREATE_BROKER:
				setBroker(createBroker(super.getName()+"_"));

				//Create VMs and Cloudlets and send them to broker
				setVmList(createVM(getBroker().getId(), 5, 100)); //creating 5 vms
				setCloudletList(createCloudlet(getBroker().getId(), 10, 100)); // creating
10 cloudlets
```

```
                                broker.submitVmList(getVmList());
                                broker.submitCloudletList(getCloudletList());

                                CloudSim.resumeSimulation();

                                break;

                        default:
                                Log.printLine(getName() + ": unknown event type");
                                break;
                        }
                }

                @Override
                public void startEntity() {
                        Log.printLine(super.getName()+" is starting...");
                        schedule(getId(), 200, CREATE_BROKER);
                }

                @Override
                public void shutdownEntity() {
                }

                public List<Vm> getVmList() {
                        return vmList;
                }

                protected void setVmList(List<Vm> vmList) {
                        this.vmList = vmList;
                }

                public List<Cloudlet> getCloudletList() {
                        return cloudletList;
                }

                protected void setCloudletList(List<Cloudlet> cloudletList) {
                        this.cloudletList = cloudletList;
                }

                public DatacenterBroker getBroker() {
                        return broker;
                }

                protected void setBroker(DatacenterBroker broker) {
                        this.broker = broker;
                }

        }

}
```

CloudSim Example 8 demonstrates how the cloudsim simulation can be paused in between as well as how new simulation entities can be created dynamically using a new global broker. For this purpose a new subclass extending SimEntity has been created and a new set of virtual machines and cloudlets are created to assign them to existing data centers.

Import Section

Two new imports are added to this example, in addition to already imported in example 7

```
import org.cloudbus.cloudsim.core.SimEntity;
import org.cloudbus.cloudsim.core.SimEvent;
```

Class Components

Same as Example 1.

Class Methods

Same as Example 6.

Important Description

Main method is similar to example 7, with minor changes. New additions are done in step 1 and 9 (as highlighted below), and major change is step 8 defined in 'example 7' has been omitted in this example. Except that every step is same

Steps	Description	Program Instructions
1	Initialize CloudSim package with number of user specification, time and trace status	**int** num_user = 2; Calendar calendar = Calendar.*getInstance*(); **boolean** trace_flag = **false**; CloudSim.*init*(num_user, calendar, trace_flag); GlobalBroker globalBroker = new GlobalBroker("GlobalBroker");
10	getCloudletReceivedList() method return a multi-parameter list related to final status of cloudlets. This will be further used by printCloudletList() method. Also Status of cloudlets executed using global broker is also added to 'newlist'.	List<Cloudlet> newList = broker.getCloudletReceivedList(); newList.addAll(globalBroker.getBroker().getCloudletReceivedList());

One nested class titled 'GlobalBroker' is explained further

Example 8- Nested Class Components

GlobalBroker is a nested class within CloudSimExample8 class and it extends SimEntity(abstract) class directly. This class is created to submit a new broker with its own pair of Virtual machines and Cloudlets. All these new VMs and CloudLets will be executed on existing DataCenters of the simulation. This case resembles where a request is being sent by a broker existing outside the current infrastructure.

Variables

There are four private variables in this class as follows:

CREATE_BROKER: Declared with private, final and static modifier, variable is used to resemble the event to create a new broker. It is been set to 0(zero) and as this variable is final so value cannot be changed on runtime.

```
"private static final int CREATE_BROKER = 0;"
```

cloudletlist: Declared with a private access modifier, cloudletlist is List data structure of type "Cloudlet" class.(i.e.) it will store objects of Cloudlet class

```
"private static List<Cloudlet> cloudletList;"
```

vmlist: Declared with a private access modifier, vmlist is a list data structure of type "VM" class. (i.e.) it will store objects of VM class.

```
"private static List<Vm> vmlist;"
```

broker: Declared with a private access modifier, broker is an object instance of DataCenterBroker Class (i.e.) it will store details regarding new global broker created.

```
"private DatacenterBroker broker;"
```

Sub -Class Methods

GlobalBroker method: This is a constructor method of the class and it sends a call to constructor of the inherited class to start adding of an entity in the simulation.

```
"public GlobalBroker(String name)"
```

processEvent method: This is a public method which is overridden definition of abstract method defined in SimEntity class works on the status of SimEvent instance. This method executes the code to create a new broker, virtual machines and Cloudlets. Once broker is created the Virtual machine list and cloudlet list is submitted to broker. To create all the above specified entities this subclass used the createBroker, createCloudlet and createVM method of its parent class. This works in the similar manner how we did in main method.

```
public processEvent(SimEvent ev)
{
switch (ev.getTag()) {
        case CREATE_BROKER:
                setBroker(createBroker(super.getName()+"_"));
                //Create VMs and Cloudlets and send them to broker
                setVmList(createVM(getBroker().getId(), 5, 100)); //creating 5 vms
                setCloudletList(createCloudlet(getBroker().getId(), 10, 100)); // creating 10 cloudlets

                broker.submitVmList(getVmList());
                broker.submitCloudletList(getCloudletList());

                //Once broker submission is done simulation is   resumed again
                CloudSim.resumeSimulation();
                break;

        default:
                Log.printLine(getName() + ": unknown event type");
                break;
        }
}
```

startEntity method: This method is a public method which is again an overridden definition of abstract method defined in SimEntity Class and this method schedules the new broker to execute task with a delay of 200 milliseconds.

"public void startEntity()"

shutdown method: This is a public & an empty method which is just declared to avoid a compilation error as this is an abstract method in SimEntity class.

"public void shutdownEntity()"

getVmList method: This public method returns existing list of VMs created in global broker class.

"public List<Vm> getVmList()"

setVmList method: This protected method is used to assign a list of VM list to the class instance.

"protected void setVmList(List<Vm> vmList)"

getCloudletList method: This public method returns existing list of cloudlets created in global broker class.

"public List<Cloudlet> getCloudletList()"

setCloudletList method: This protected method used to assign a list of cloudlets list to the class instance.

*"**protected void** setCloudletList(List<Cloudlet> cloudletList)"*

getBroker method: This public method is used to return the instance of broker created through GlobalBroker Class.

*"**public** DatacenterBroker getBroker()"*

setBroker method: This protected method is used to assign the external instance of broker to broker instance of the GlobalBroker class.

*"**protected void** setBroker(DatacenterBroker broker)"*