**1.How do you design an application with JMS messaging?**

Below are the steps to design JMS application

Step 1

Look up a connection factory in JNDI.

qconFactory = (QueueConnectionFactory) ctx.lookup(JMS\_FACTORY);

Step 2

Create a connection using the connection factory.

qcon = qconFactory.createQueueConnection();

Step 3

Create a session using the connection. The following code defines the session as non-transacted and specifies that messages will be acknowledged automatically. For more information about transacted sessions and acknowledge modes, see Session.

qsession = qcon.createQueueSession(false, Session.AUTO\_ACKNOWLEDGE);

Step 4

Look up a destination (queue) in JNDI.

queue = (Queue) ctx.lookup(queueName);

Step 5

Create a reference to a message producer (queue sender) using the session and destination (queue).

qsender = qsession.createSender(queue);

Step 6

Create the message object.

msg = qsession.createTextMessage();

Step 7

Start the connection.

qcon.start(); }

The init() method for the examples.jms.queue.QueueReceive example is similar to the QueueSend init() method shown previously, with the one exception. Steps 5 and 6 would be replaced by the following code, respectively:

qreceiver = qsession.createReceiver(queue); qreceiver.setMessageListener(this);

In the first line, instead of calling the createSender() method to create a reference to the queue sender, the application calls the createReceiver() method to create the queue receiver.

In the second line, the message consumer registers an asynchronous message listener.

When a message is delivered to the queue session, it is passed to the examples.jms.QueueReceive.onMessage()method.

**2. How do you handle exception in JMS consumers and how to you recover?**

You should create the session like this:

Session session = connection.createSession(false, Session.CLIENT\_ACKNOWLEDGE);

when you try to deliver the message to your third-party app:

If it's working you should acknoledge the message.

If it is down you should'nt acknwoledge it, this way the JMS provider will be able to rediliver it,and the message will not be lost. message.acknowledge();

This can be acheived my using session acknowledgement. For this first modify your producer code to use Session.AUTO\_ACKNOWLEDGE. While creating Queue session, make AUTO\_ACKNOWLEDGE as false. That means consumer must acknowledge. When the consumer sends acknowledgement of message, then the message will be deleted from the queue, otherwise it will remain in the queue.

Below is the producer code.

try { QueueConnectionFactory qcf = AppUtils.getQueueConnectionFactory(); Queue q = AppUtils.getDestination(); QueueConnection qConnection = qcf.createQueueConnection(); QueueSession qSession = qConnection.createQueueSession(false, Session.AUTO\_ACKNOWLEDGE); QueueSender qSender = qSession.createSender(q); qConnection.start(); TextMessage msg = qSession.createTextMessage("Hello"); qSender.send(msg); qSender.close(); qConnection.close(); } catch (JMSException e) { // log your error to log file e.printStackTrace(); }

On the consumer side, you must do the same thing, create a queue session with AUTO\_ACKNOWLEDGE as false.

After working on your message, you can send acknowledge to delete the message from the queue or the message will remain in the queue.

try { QueueConnectionFactory qcf = getQueueConnectionFactory(); Queue q = getDestination(); QueueConnection qConnection = qcf.createQueueConnection(); QueueSession qSession = qConnection.createQueueSession(false, Session.AUTO\_ACKNOWLEDGE); QueueReceiver qReceiver = qSession.createReceiver(q); qConnection.start(); Message msg = qReceiver.receive(); // here send your message to third party application //if your third party application is down if(thirdpartyapp is down){ //here you can raise an exception //or just do nothing // you're not sending acknowledgement here so the msg will //remain in the queue }else{ msg.acknowledge();//here youre sending ack, so msg will be deleted qReceiver.close(); qConnection.close(); } } catch (JMSException e) { // TODO Auto-generated catch block e.printStackTrace(); }

**3. How do you implement LRU or MRU cache?**

Use doubly linked list with a hash function.

Hash function helps locate a node.

Double LL is needed when a node is accessed and need to be moved to head of LL(recently used.)

when elements in LL > threshold. remove element at end of lru

LRU Cache:

In computing, cache replacement algorithms are optimizing algorithms that a computer program or a hardware maintained structure can follow to manage a cache of information stored on the computer. When the cache is full, the algorithm must choose which items to discard to make room for the new ones.

Initially, when the cache has room for more pages, we keep on adding the pages to the cache. But when cache is filled, then to add a new page, another page must be removed. So, a strategy needs to be used for replacing cache pages.

Least Recently Used cache replacement algorithm is a cache replacement strategy by which the least recently accessed page is removed from the cache when a new page is accessed which is not already present in the cache.

For implementing an LRU cache, we can use a doubly linked list and a hash map.

Doubly Linked List - List of pages with most recently used page at the start of the list. So, as more pages are added to the list, least recently used pages are moved to the end of the list with page at tail being the least recently used page in the list.

Hash Map (key: page number, value: page) - For O(1) access to pages in cache

When a page is accessed, there can be 2 cases:

1. Page is present in the cache - If the page is already present in the cache, we move the page to the start of the list.

2. Page is not present in the cache - If the page is not present in the cache, we add the page to the list.

How to add a page to the list:

   a. If the cache is not full, add the new page to the start of the list.

   b. If the cache is full, remove the last node of the linked list and move the new page to the start of the list.

Java code is provided in code snippet section.

Please refer to Algorithm Visualization section for understanding how the algorithm works for different test cases.

Code Snippet

package com.ideserve.questions.excel;

import java.util.HashMap;

import java.util.Map;

public class LRUCache

{

   private DoublyLinkedList pageList;

   private Map<Integer, Node> pageMap;

   private final int cacheSize;

   public LRUCache(int cacheSize) {

    this.cacheSize = cacheSize;

    pageList = new DoublyLinkedList(cacheSize);

    pageMap = new HashMap<Integer, Node>();

    }

    public void accessPage(int pageNumber)

{

        Node pageNode = null;

        if(pageMap.containsKey(pageNumber)) {

       // If page is present in the cache, move the page to the start of list

            pageNode = pageMap.get(pageNumber);

            pageList.movePageToHead(pageNode);

        }

else {

            // If the page is not present in the cache, add the page to the cache

            if(pageList.getCurrSize() == pageList.getSize()) {

                // If cache is full, we will remove the tail from the cache pageList

                // Remove it from map too

                pageMap.remove(pageList.getTail().getPageNumber());

            }

pageNode = pageList.addPageToList(pageNumber);

            pageMap.put(pageNumber, pageNode);

        }

    }

    public void printCacheState()

{

       pageList.printList();

        System.out.println();

    }

    public static void main(String[] args)

{

      int cacheSize = 4;

        LRUCache cache = new LRUCache(cacheSize);

        cache.accessPage(4);

        cache.printCacheState();

        cache.accessPage(2);

        cache.printCacheState();

        cache.accessPage(1);

        cache.printCacheState();

        cache.accessPage(1);

        cache.printCacheState();

        cache.accessPage(4);

        cache.printCacheState();

        cache.accessPage(3);

        cache.printCacheState();

        cache.accessPage(7);

        cache.printCacheState();

        cache.accessPage(8);

        cache.printCacheState();

        cache.accessPage(3);

        cache.printCacheState();

    }

}

class DoublyLinkedList

{

   private final int size;

    private int currSize;

    private Node head;

    private Node tail;

    public DoublyLinkedList(int size)

{

       this.size = size;

        currSize = 0;

    }

    public Node getTail()

{

        return tail;

    }

    public void printList()

{

       if(head == null)

{

           return;

        }

        Node tmp = head;

while(tmp != null)

{

            System.out.print(tmp);

            tmp = tmp.getNext();

        }

    }

    public Node addPageToList(int pageNumber)

{

        Node pageNode = new Node(pageNumber);

        if(head == null)

{

            head = pageNode;

            tail = pageNode;

            currSize = 1;

            return pageNode;

        }

else if(currSize < size)

{

            currSize++;

        }

else

{

            tail = tail.getPrev();

            tail.setNext(null);

        }

        pageNode.setNext(head);

        head.setPrev(pageNode);

        head = pageNode;

        return pageNode;

    }

    public void movePageToHead(Node pageNode)

{

        if(pageNode == null || pageNode == head) {

            return;

        }

        if(pageNode == tail) {

            tail = tail.getPrev();

            tail.setNext(null);

        }

        Node prev = pageNode.getPrev();

        Node next = pageNode.getNext();

        prev.setNext(next);

        if(next != null) {

            next.setPrev(prev);

        }

        pageNode.setPrev(null);

        pageNode.setNext(head);

        head.setPrev(pageNode);

        head = pageNode;

    }

    public int getCurrSize()

{

        return currSize;

    }

   public void setCurrSize(int currSize)

{

        this.currSize = currSize;

    }

   public Node getHead()

{

       return head;

    }

    public void setHead(Node head)

{

       this.head = head;

    }

    public int getSize()

{

        return size;

    }

}

class Node {

   private int pageNumber;

    private Node prev;

    private Node next;

public Node(int pageNumber)

{

       this.pageNumber = pageNumber;

    }

    public int getPageNumber()

{

        return pageNumber;

    }

   public void setPageNumber(int data)

{

       this.pageNumber = data;

    }

    public Node getPrev()

{

      return prev;

    }

    public void setPrev(Node prev)

{

       this.prev = prev;

    }

   public Node getNext()

{

       return next;

    }

    public void setNext(Node next)

{

        this.next = next;

    }

   public String toString()

{

        return pageNumber + "  ";

    }

}

**4. How would you implement Executor Service?**

Executor Service is an interface that extends Executor class and represents an asynchronous execution. It provides us mechanisms to manage the end and detect progress of the asynchronous tasks.

In this example, we are going to see some basic functionalities of ExecutorService, as well as handle the Future object, the result of asynchronous computation.

1. Create the Runnable

We are going to create a Runnable that is intended to be executed by the ExecutorService. Create a java class named myThread and paste the following code.

*myThread.java:*

|  |  |
| --- | --- |
| package com.javacodegeeks.core.concurrency.executorservicetest; | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | public class MyThread implements Runnable { |
| 04 |  |

|  |  |
| --- | --- |
| 05 | private String myName; |
| 06 | private int count; |

|  |  |
| --- | --- |
| 07 | private final long timeSleep; |
| 08 |  |

|  |  |
| --- | --- |
| 09 | MyThread(String name, int newcount, long newtimeSleep) { |
| 10 | this.myName = name; |

|  |  |
| --- | --- |
| 11 | this.count = newcount; |
| 12 | this.timeSleep = newtimeSleep; |

|  |  |
| --- | --- |
| 13 | } |
| 14 |  |

|  |  |
| --- | --- |
| 15 | @Override |
| 16 | public void run() { |

|  |  |
| --- | --- |
| 17 | // TODO Auto-generated method stub |
| 18 |  |

|  |  |
| --- | --- |
| 19 | int sum = 0; |
| 20 | for (int i = 1; i <= this.count; i++) { |

|  |  |
| --- | --- |
| 21 | sum = sum + i; |
| 22 | } |

|  |  |
| --- | --- |
| 23 | System.out.println(myName + " thread has sum = " + sum + |
| 24 | " and is going to sleep for " + timeSleep); |

|  |  |
| --- | --- |
| 25 | try { |
| 26 | Thread.sleep(this.timeSleep); |

|  |  |
| --- | --- |
| 27 | } catch (InterruptedException e) { |
| 28 | // TODO Auto-generated catch block |

|  |  |
| --- | --- |
| 29 | e.printStackTrace(); |
| 30 | } |

|  |  |
| --- | --- |
| 31 | } |
| 32 |  |

|  |  |
| --- | --- |
| 33 | } |

The functionality of the Runnable is very simple. It computes a sum from the giving argument and it sleeps for a specified time.

2. Code the ExecutorService

In this example we will use a factor method of ExecutorService that creates a thread pool of fixed number of threads. For this reason, newFixedThreadPool() method is used where we specify the number of threads in the pool. To execute the thread, we can use either execute() method or submit(), where both of them take Runnable as a parameter. execute()method is depending on the implementation of the Executor class and may perform the Runnable in a new thread, in a pooled thread, or in the calling thread. submit() method extends execute(), by returning a Future that represents the submitting task.

The Future can be used to indicate the termination of execution of the thread. For instance, get() method waits for the completion of the computation. If the returning value is null, the task has finished correctly. Otherwise, cancel() method can be called in order to end the execution of this task. It is worth to mention that for bulk or a collection of thread execution, invokeAll() and invokeAny() are used respectively, although there are not used in this example.

To close down the ExecutorService, there are many methods that can be used. In our example we use shutdown()method, in which the submitted tasks are executed before the shutting down but new tasks can not be accepted. Another approach is shutdownNow() method, which stops the executing tasks, pause the waiting ones and returns the list of the awaiting ones. Moreover, awaitTermination() can be used in order to wait until all threads are terminated.

For further understanding of the main functionality of ExecutorService, have a look at the code below. Create ExecutorServiceTest.java file and paste the following.

*ExecutorServiceTest.java:*

|  |  |
| --- | --- |
| 01 | package com.javacodegeeks.core.concurrency.executorservicetest; |
| 02 |  |

|  |  |
| --- | --- |
| 03 | import java.util.concurrent.ExecutionException; |
| 04 | import java.util.concurrent.ExecutorService; |

|  |  |
| --- | --- |
| 05 | import java.util.concurrent.Executors; |
| 06 | import java.util.concurrent.Future; |

|  |  |
| --- | --- |
| 07 | import java.util.concurrent.TimeUnit; |
| 08 |  |

|  |  |
| --- | --- |
| 09 | public class ExecutorServiceTest { |
| 10 |  |

|  |  |
| --- | --- |
| 11 | private static Future taskTwo = null; |
| 12 | private static Future taskThree = null; |

|  |  |
| --- | --- |
| 13 |  |
| 14 | public static void main(String[] args) throws InterruptedException, ExecutionException { |

|  |  |
| --- | --- |
| 15 | ExecutorService executor = Executors.newFixedThreadPool(2); |
| 16 |  |

|  |  |
| --- | --- |
| 17 | // execute the Runnable |
| 18 | Runnable taskOne = new MyThread("TaskOne", 2, 100); |

|  |  |
| --- | --- |
| 19 | executor.execute(taskOne); |
| 20 | for(int i = 0; i < 2; i++) { |

|  |  |
| --- | --- |
| 21 | // if this task is not created or is canceled or is completed |
| 22 | if ((taskTwo == null) || (taskTwo.isDone()) || (taskTwo.isCancelled())) { |

|  |  |
| --- | --- |
| 23 | // submit a task and return a Future |
| 24 | taskTwo = executor.submit(new MyThread("TaskTwo", 4, 200)); |

|  |  |
| --- | --- |
| 25 | } |
| 26 |  |

|  |  |
| --- | --- |
| 27 | if ((taskThree == null) || (taskThree.isDone()) || (taskThree.isCancelled())) { |
| 28 | taskThree = executor.submit(new MyThread("TaskThree", 5, 100)); |

|  |  |
| --- | --- |
| 29 | } |
| 30 | // if null the task has finished |

|  |  |
| --- | --- |
| 31 | if(taskTwo.get() == null) { |
| 32 | System.out.println(i+1 + ") TaskTwo terminated successfully"); |

|  |  |
| --- | --- |
| 33 | } else { |
| 34 | // if it doesn't finished, cancel it |

|  |  |
| --- | --- |
| 35 | taskTwo.cancel(true); |
| 36 | } |

|  |  |
| --- | --- |
| 37 | if(taskThree.get() == null) { |
| 38 | System.out.println(i+1 + ") TaskThree terminated successfully"); |

|  |  |
| --- | --- |
| 39 | } else { |
| 40 | taskThree.cancel(true); |

|  |  |
| --- | --- |
| 41 | } |
| 42 | } |

|  |  |
| --- | --- |
| 43 | executor.shutdown(); |
| 44 | System.out.println("-----------------------"); |

|  |  |
| --- | --- |
| 45 | // wait until all tasks are finished |
| 46 | executor.awaitTermination(1, TimeUnit.SECONDS); |

|  |  |
| --- | --- |
| 47 | System.out.println("All tasks are finished!"); |
| 48 |  |

|  |  |
| --- | --- |
| 49 | } |
| 50 |  |

|  |  |
| --- | --- |
| 51 | } |

Now you can see the output of the execution.

*Output:*

TaskOne thread has sum = 3 and is going to sleep for 100

TaskTwo thread has sum = 10 and is going to sleep for 200

TaskThree thread has sum = 15 and is going to sleep for 100

1) TaskTwo terminated successfully

1) TaskThree terminated successfully

TaskTwo thread has sum = 10 and is going to sleep for 200

TaskThree thread has sum = 15 and is going to sleep for 100

2) TaskTwo terminated successfully

2) TaskThree terminated successfully

-----------------------

All tasks are finished!

**4. Describe singleton design pattern – how would you implement?**

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Implementation

We're going to create a *SingleObject* class. *SingleObject* class have its constructor as private and have a static instance of itself.

*SingleObject* class provides a static method to get its static instance to outside world. *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.

Create a Singleton Class.

*SingleObject.java*

public class SingleObject {

//create an object of SingleObject

private static SingleObject instance = new SingleObject();

//make the constructor private so that this class cannot be

//instantiated

private SingleObject(){}

//Get the only object available

public static SingleObject getInstance(){

return instance;

}

public void showMessage(){

System.out.println("Hello World!");

}

}

Get the only object from the singleton class.

*SingletonPatternDemo.java*

public class SingletonPatternDemo {

public static void main(String[] args) {

//illegal construct

//Compile Time Error: The constructor SingleObject() is not visible

//SingleObject object = new SingleObject();

//Get the only object available

SingleObject object = SingleObject.getInstance();

//show the message

object.showMessage();

}

}

**5. Describe properties of Java String.**

A Java String contains an immutable sequence of Unicode characters. Unlike C/C++, where string is simply an array of char, A Java String is an object of the class java.lang.

Java String is, however, special. Unlike an ordinary class:

* String is associated with string literal in the form of double-quoted texts such as "Hello, world!". You can assign a string literal directly into a String variable, instead of calling the constructor to create a String instance.
* The '+' operator is overloaded to concatenate two String operands. '+' does not work on any other objects such as Point and Circle.
* String is *immutable*. That is, its content cannot be modified once it is created. For example, the method toUpperCase() constructs and returns a new String instead of modifying the its existing content.

Strings receive *special treatment* in Java, because they are used frequently in a program. Hence, efficiency (in terms of computation and storage) is crucial.

The designers of Java decided to retain primitive types in an object-oriented language, instead of making everything an object, so as to improve the performance of the language. Primitives are stored in the call stack, which require less storage spaces and are cheaper to manipulate. On the other hand, objects are stored in the program heap, which require complex memory management and more storage spaces.

For performance reason, Java's String is designed to be in between a primitive and a class. The special features in String include:

* The '+' operator, which performs addition on primitives (such as int and double), is overloaded to operate on String objects. '+' performs concatenation for two String operands.  
  Java does not support *operator overloading* for software engineering consideration. In a language that supports operator overloading like C++, you can turn a '+' operator to perform a subtraction, resulted in poor codes. The '+' operator is the *only* operator that is internally overloaded to support string concatenation in Java. Take note that '+' does not work on any two arbitrary objects, such as Points or Circles.
* A String can be constructed by either:
  1. directly assigning a string *literal* to a String reference - *just like a primitive*, or
  2. via the "new" operator and constructor, similar to any other classes. However, this is not commonly-used and is not recommended
* String literals are stored in *a common pool*. This facilitates *sharing of storage* for strings with the same contents to conserve storage. String objects allocated via new operator are stored in the heap, and there is no sharing of storage for the same contents.