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

Step 1: Look Up a Connection Factory in JNDI

Once the connection factory has been defined, you can look it up by first establishing a JNDI context (context) using the InitialContext() method. For any application other than a servlet application, you must pass an environment used to create the initial context.

QueueConnectionFactory queueConnectionFactory =

(QueueConnectionFactory) context.lookup(CF\_name);

TopicConnectionFactory topicConnectionFactory =

(TopicConnectionFactory) context.lookup(CF\_name);

Step 2: Create a Connection Using the Connection Factory

Create a Queue Connection

The QueueConnectionFactory provides the following two methods for creating a queue connection:

public QueueConnection createQueueConnection(

) throws JMSException

public QueueConnection createQueueConnection(

String userName,

String password

) throws JMSException

Step 3: Create a Session Using the Connection

You can create one or more sessions for accessing a queue or topic using the Connection methods described in the following sections.

Create a Queue Session

The QueueConnection class defines the following method for creating a queue session:

public QueueSession createQueueSession(

boolean transacted,

int acknowledgeMode

) throws JMSException

Create a Topic Session

The TopicConnection class defines the following method for creating a topic session:

public TopicSession createTopicSession(

boolean transacted,

int acknowledgeMode

) throws JMSException

Step 4: Look Up a Destination (Queue or Topic)

Using a JNDI Name

Queue queue = (Queue) context.lookup(Dest\_name);

Topic topic = (Topic) context.lookup(Dest\_name);

The Dest\_name argument specifies the destination's JNDI name defined during configuration.

.

public Queue createQueue(

String queueName

) throws JMSException

public Topic createTopic(

String topicName

) throws JMSException

public String getQueueName(

) throws JMSException

public String getTopicName(

) throws JMSException

Step 5: Create Message Producers and Message Consumers Using the Session and Destinations

You can create message producers and message consumers by passing the destination reference

Create QueueSenders and QueueReceivers

The QueueSession object defines the following methods for creating queue senders and receivers:

public QueueSender createSender(

Queue queue

) throws JMSException

public QueueReceiver createReceiver(

Queue queue

) throws JMSException

public QueueReceiver createReceiver(

Queue queue,

String messageSelector

) throws JMSException

Once the queue sender or receiver has been created, you can access the queue name associated with the queue sender or receiver using the following QueueSender or QueueReceiver method:

public Queue getQueue(

) throws JMSException

public TopicPublisher createPublisher(

Topic topic

) throws JMSException

public TopicSubscriber createSubscriber(

Topic topic

) throws JMSException

public TopicSubscriber createSubscriber(

Topic topic,

String messageSelector,

boolean noLocal

) throws JMSException

Step 6a: Create the Message Object (Message Producers)

Session Methods

Note: These methods are inherited by both the QueueSession and TopicSession subclasses.

public BytesMessage createBytesMessage(

) throws JMSException

public MapMessage createMapMessage(

) throws JMSException

public Message createMessage(

) throws JMSException

public ObjectMessage createObjectMessage(

) throws JMSException

public ObjectMessage createObjectMessage(

Serializable object

) throws JMSException

public StreamMessage createStreamMessage(

) throws JMSException

public TextMessage createTextMessage(

) throws JMSException

public TextMessage createTextMessage(

String text

) throws JMSException

WLSession Method

public XMLMessage createXMLMessage(

String text

) throws JMSException

Step 6b: Optionally Register an Asynchronous Message Listener (Message Consumers)

Implement the javax.jms.MessageListener interface, which includes an onMessage() method.

public void setMessageListener(

MessageListener listener

) throws JMSException

MessageConsumer method:

public MessageListener getMessageListener(

) throws JMSException

Step 7: Start the Connection

You start the connection using the Connection class start() 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.

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

Implement LRU Cache

How to implement LRU caching scheme? What data structures should be used?

We are given total possible page numbers that can be referred. We are also given cache (or memory) size (Number of page frames that cache can hold at a time). The LRU caching scheme is to remove the least recently used frame when the cache is full and a new page is referenced which is not there in cache. Please see the Galvin book for more details

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

Code Snippet

public class LRUCache<K, V>{

// Define Node with pointers to the previous and next items and a key, value pair

class Node<T, U> {

Node<T, U> previous;

Node<T, U> next;

T key;

U value;

public Node(Node<T, U> previous, Node<T, U> next, T key, U value){

this.previous = previous;

this.next = next;

this.key = key;

this.value = value;

}

}

private HashMap<K, Node<K, V>> cache;

private Node<K, V> leastRecentlyUsed;

private Node<K, V> mostRecentlyUsed;

private int maxSize;

private int currentSize;

public LRUCache(int maxSize){

this.maxSize = maxSize;

this.currentSize = 0;

leastRecentlyUsed = new Node<K, V>(null, null, null, null);

mostRecentlyUsed = leastRecentlyUsed;

cache = new HashMap<K, Node<K, V>>();

}

public V get(K key){

Node<K, V> tempNode = cache.get(key);

if (tempNode == null){

return null;

}

// If MRU leave the list as it is

else if (tempNode.key == mostRecentlyUsed.key){

return mostRecentlyUsed.value;

}

// Get the next and previous nodes

Node<K, V> nextNode = tempNode.next;

Node<K, V> previousNode = tempNode.previous;

// If at the left-most, we update LRU

if (tempNode.key == leastRecentlyUsed.key){

nextNode.previous = null;

leastRecentlyUsed = nextNode;

}

// If we are in the middle, we need to update the items before and after our item

else if (tempNode.key != mostRecentlyUsed.key){

previousNode.next = nextNode;

nextNode.previous = previousNode;

}

// Finally move our item to the MRU

tempNode.previous = mostRecentlyUsed;

mostRecentlyUsed.next = tempNode;

mostRecentlyUsed = tempNode;

mostRecentlyUsed.next = null;

return tempNode.value;

}

public void put(K key, V value){

if (cache.containsKey(key)){

return;

}

// Put the new node at the right-most end of the linked-list

Node<K, V> myNode = new Node<K, V>(mostRecentlyUsed, null, key, value);

mostRecentlyUsed.next = myNode;

cache.put(key, myNode);

mostRecentlyUsed = myNode;

// Delete the left-most entry and update the LRU pointer

if (currentSize == maxSize){

cache.remove(leastRecentlyUsed.key);

leastRecentlyUsed = leastRecentlyUsed.next;

leastRecentlyUsed.previous = null;

}

// Update cache size, for the first added entry update the LRU pointer

else if (currentSize < maxSize){

if (currentSize == 0){

leastRecentlyUsed = myNode;

}

currentSize++;

}

}

}

Reference-🡪

**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.

## ExecutorService Example

Here is a simple Java ExectorService example:

ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

First an ExecutorService is created using the newFixedThreadPool() factory method. This creates a thread pool with 10 threads executing tasks.

Second, an anonymous implementation of the Runnable interface is passed to the execute() method. This causes the Runnable to be executed by one of the threads in the ExecutorService.

## Task Delegation

Here is a diagram illustrating a thread delegating a task to an ExecutorService for asynchronous execution:

|  |
| --- |
| A thread delegating a task to an ExecutorService for asynchronous execution. |
| **A thread delegating a task to an ExecutorService for asynchronous execution.** |

Once the thread has delegated the task to the ExecutorService, the thread continues its own execution independent of the execution of that task.

## ExecutorService Implementations

Since ExecutorService is an interface, you need to its implementations in order to make any use of it. The ExecutorService has the following implementation in the java.util.concurrent package:

* [**ThreadPoolExecutor**](http://tutorials.jenkov.com/java-util-concurrent/threadpoolexecutor.html)
* [**ScheduledThreadPoolExecutor**](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html)

## Creating an ExecutorService

How you create an ExecutorService depends on the implementation you use. However, you can use the Executors factory class to create ExecutorService instances too. Here are a few examples of creating an ExecutorService:

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

## ExecutorService Usage

There are a few different ways to delegate tasks for execution to an ExecutorService:

* execute(Runnable)
* submit(Runnable)
* submit(Callable)
* invokeAny(...)
* invokeAll(...)

I will take a look at each of these methods in the following sections.

### execute(Runnable)

The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously. Here is an example of executing a Runnable with an ExecutorService:

ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

There is no way of obtaining the result of the executed Runnable, if necessary. You will have to use a Callable for that (explained in the following sections).

### submit(Runnable)

The submit(Runnable) method also takes a Runnable implementation, but returns a Future object. This Future object can be used to check if the Runnable as finished executing.

Here is a ExecutorService submit() example:

Future future = executorService.submit(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

future.get(); //returns null if the task has finished correctly.

### submit(Callable)

The submit(Callable) method is similar to the submit(Runnable) method except for the type of parameter it takes. The Callable instance is very similar to a Runnable except that its call() method can return a result. The Runnable.run() method cannot return a result.

The Callable's result can be obtained via the Future object returned by the submit(Callable) method. Here is an ExecutorService Callable example:

Future future = executorService.submit(new Callable(){

public Object call() throws Exception {

System.out.println("Asynchronous Callable");

return "Callable Result";

}

});

System.out.println("future.get() = " + future.get());

The above code example will output this:

Asynchronous Callable

future.get() = Callable Result

### invokeAny()

The invokeAny() method takes a collection of Callable objects, or subinterfaces of Callable. Invoking this method does not return a Future, but returns the result of one of the Callable objects. You have no guarantee about which of the Callable's results you get. Just one of the ones that finish.

If one of the tasks complete (or throws an exception), the rest of the Callable's are cancelled.

Here is a code example:

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

String result = executorService.invokeAny(callables);

System.out.println("result = " + result);

executorService.shutdown();

This code example will print out the object returned by one of the Callable's in the given collection. I have tried running it a few times, and the result changes. Sometimes it is "Task 1", sometimes "Task 2" etc.

### invokeAll()

The invokeAll() method invokes all of the Callable objects you pass to it in the collection passed as parameter. The invokeAll() returns a list of Future objects via which you can obtain the results of the executions of each Callable.

Keep in mind that a task might finish due to an exception, so it may not have "succeeded". There is no way on a Future to tell the difference.

Here is a code example:

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){

System.out.println("future.get = " + future.get());

}

executorService.shutdown();

## ExecutorService Shutdown

When you are done using the ExecutorService you should shut it down, so the threads do not keep running.

For instance, if your application is started via a main() method and your main thread exits your application, the application will keep running if you have an active ExexutorService in your application. The active threads inside this ExecutorService prevents the JVM from shutting down.

To terminate the threads inside the ExecutorService you call its shutdown() method. The ExecutorServicewill not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the ExecutorService shuts down. All tasks submitted to the ExecutorService before shutdown() is called, are executed.

If you want to shut down the ExecutorService immediately, you can call the shutdownNow() method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks. There are no guarantees given about the executing tasks. Perhaps they stop, perhaps the execute until the end. It is a best effort attempt.

**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.