NFP121

Programmation avancée.

Session 1, 31 janvier 2017 - durée : 3 heures

Tous documents papiers autorisés

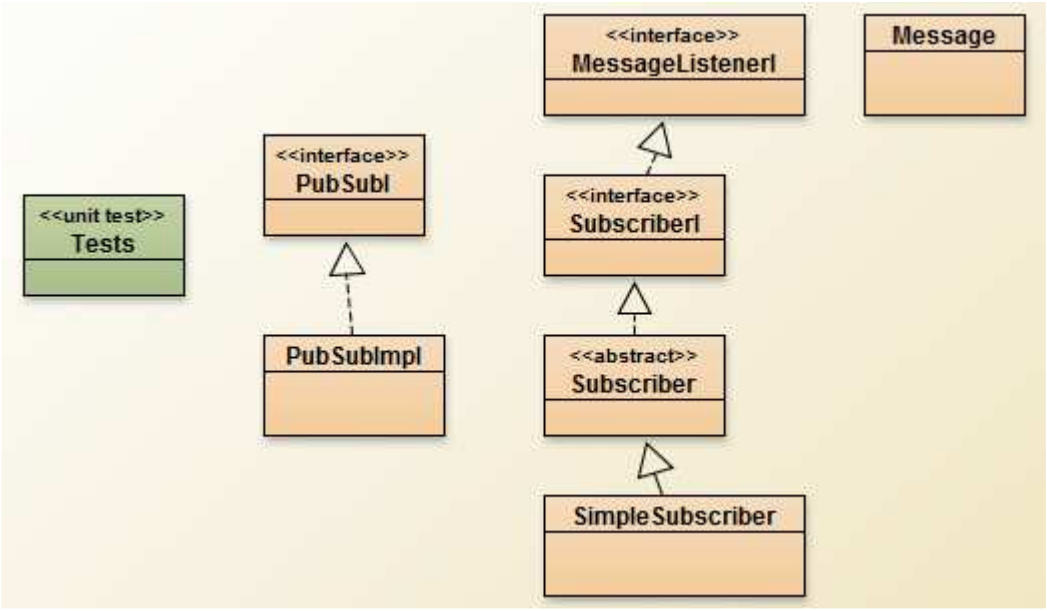
Cnam / Paris-HTO & FOD/Nationale

Sommaire :

Question1 : Patron *PubSub*

L'usage de ce patron permet à un souscripteur, d'être notifié à la publication d'un message. La publication s'effectue en précisant le nom du ou des souscripteurs, et le message envoyé contient le nom du publieur et un texte.

Ci-dessous, en syntaxe UML/BlueJ, une architecture des classes:



Cette architecture est constituée de l'interface ***PubSubI***, d'une implémentation ***PubSubImpl***. L'interface ***MessageListener*** contient l'entête de la méthode qui sera déclenchée à chaque notification. La classe ***SimpleSubscriber*** se contente d'afficher le message reçu sur la console, cette classe hérite de la classe abstraite ***Subscriber***. Une classe de tests unitaires ***Tests*** est fournie.

L'interface ***PubSubI*** ci-dessous**,** propose les opérations :

- de souscription (`subscribe`),
- de publication d'un message à un seul souscripteur (`publish`),
- de publication d'un message à plusieurs souscripteurs (`publish`),
- d'obtention d'un itérateur sur les souscripteurs inscrits (`iterator`).

```
public interface PubSubI extends Iterable<SubscriberI>, Serializable{

  /** Publication d'un message à  certains souscripteurs.
   * @param names les noms des souscripteurs
   * @param message le message à  transmettre
   * @return le nombre de souscripteurs ayant reçu le message
   */
  public int publish(String[] names, Message message);

  /** Publication d'un message à un souscripteur.
   * @param name le nom du souscripteur
   * @param message le message à  transmettre
   * @return true si l'envoi est réussi, false autrement
   */
  public boolean publish(String name, Message message);

  /** Souscription.
   * @param subscriber le souscripteur
   * @return false si ce souscripteur a déjà souscrit avec le même nom
   */
  public boolean subscribe(SubscriberI subscriber);

  /** Obtention d'un itérateur.
   * @return un itérateur sur les souscripteurs enregistrés
   */
  public Iterator<SubscriberI> iterator();
}
```

L'interface ***SubscriberI***, suivie de ***MessageListenerI*** :

```
public interface SubscriberI extends MessageListenerI, Serializable{
  /** Obtention du nom du souscripteur.
   * @return le nom du souscripteur
   */
  public String getName();

  /** Obtention de la liste des abonnements de ce souscripteur.
   * @return la liste des souscriptions effectuées
   */
  public List<PubSubI> getPubSubList();

  /** Ajout d'un abonnement pour ce souscripteur.
   * @return false si l'abonnement était déjà en place
   */
  public boolean addPubSub(PubSubI pubsub);

}

public interface MessageListenerI{
  /** Méthode déclenchée à  chaque réception d'un message par un souscripteur.
   * @param message le message reçu.
   * @throws Exception levée par le destinataire, i.e. une erreur lors de la réception du message
   */
  public void onMessage(Message message) throws Exception;
}
```

Les classes ***Message*** et ***SimpleSubscriber*** :

```
public class Message{
  private String source;
  private String content;

  public Message(String source, String content){
    this.source = source;
    this.content = content;
  }
```

```
  public String getSource(){return source; }

  public String getContent(){return content; }

  public String toString(){
    return "<" + source + ", " + content + ">";
  }
}

public class SimpleSubscriber extends Subscriber{

  public SimpleSubscriber(String name){
    super(name);
  }
  public void onMessage(Message message) throws Exception{
    System.out.println(getName() + " : " + message);
  }
}
```

Ci-dessous la classe ***Tests*** : une classe de tests unitaires, à lire attentivement avant de répondre à cette question.

```
public class Tests extends junit.framework.TestCase{

    class SubscriberTest extends SimpleSubscriber{
      Stack<Message> messages;
      SubscriberTest(PubSubI pubsub, String name){
        super(name);
        pubsub.subscribe(this);                    // souscription de cette instance
        this.messages = new Stack<Message>();      // une pile de messages
      }
      public void onMessage(Message message)throws Exception{
        this.messages.push(message); // sauvegarde du message sur la pile
                        // pour les tests, une exception est levée si le message est "error"
        if("error".equals(message.getContent()))throw new Exception();
        super.onMessage(message);
      }
    }

    public void testSimple(){
       PubSubI pubsub = new PubSubImpl("pubsub");
       SubscriberTest a = new SubscriberTest(pubsub,"a");
       assertEquals("a", a.getName());
       assertTrue(a.getPubSubList().contains(pubsub)); // "a" a bien souscrit
       assertEquals(a, pubsub.iterator().next());// "a" est bien présent

       SubscriberTest b = new SubscriberTest(pubsub,"b");
       SubscriberTest c = new SubscriberTest(pubsub,"c");

       assertFalse(pubsub.subscribe(a)); // "a" a déjà souscrit
       Message message = new Message("p1","test");

       assertEquals(2,pubsub.publish(new String[]{"a","b"}, message));
                  // 2 souscripteurs ont reçu le message, ici a et b
       assertEquals("test",a.messages.pop().getContent());
       assertEquals("test",b.messages.pop().getContent());
       assertTrue(c.messages.isEmpty());
       assertEquals(3,pubsub.publish(new String[]{"c","a","b"}, message));
       assertEquals(0,pubsub.publish(new String[]{"d","e"}, message));
                    // 0 : pas de réception, d et e n'ont pas souscrit
    }

    public void testAvecException(){
       PubSubI pubsub = new PubSubImpl("pubsub");
       SubscriberTest a = new SubscriberTest(pubsub,"a");
       SubscriberTest b = new SubscriberTest(pubsub,"b");
       Message message = new Message("p1","error");  // exception programmée
       assertEquals(0,pubsub.publish(new String[]{"a","b","c"}, message));
                  // 0 : pas de réception, "error" lève une exception
    }
```

```
    public void testDeuxPubSubUnSouscripteur(){
       PubSubI pubsub1 = new PubSubImpl("pubsub1");
       PubSubI pubsub2 = new PubSubImpl("pubsub2");
       SubscriberTest st1 = new SubscriberTest(pubsub1,"a");
       SubscriberTest st2 = new SubscriberTest(pubsub2,"a");
       assertTrue(st1.getPubSubList().contains(pubsub1));
       assertEquals("a",st1.getName());
       assertTrue(st2.getPubSubList().contains(pubsub2));
       assertEquals("a",st2.getName());
       Message message = new Message("p1","test");
       assertEquals(1,pubsub1.publish(new String[]{"a"}, message));
       assertEquals("test",st1.messages.pop().getContent());
       assertEquals(1,pubsub2.publish(new String[]{"a"}, message));
       assertEquals("test",st2.messages.pop().getContent());
    }
}
```

Question1-1)

Ecrivez une implémentation **complète des classes *Subscriber* et *PubSubImpl***. Certaines documentations d'interfaces et de classes sont en annexe. Une explication en quelques lignes précisant vos choix d'implémentations est demandée. Vous pouvez détacher une des dernières pages de cet énoncé, la joindre à votre copie en n'oubliant pas de reporter le numéro de celle-ci.

Question1-2) DurableSubscriber

Ci-dessous, un extrait de http://www.enterpriseintegrationpatterns.com/patterns/messaging/DurableSubscription.html

A durable subscription saves messages for an inactive subscriber and delivers these saved messages when the subscriber reconnects. In this way, a subscriber will not lose any messages even though it disconnected. A durable subscription has no effect on the behavior of the subscriber or the messaging system while the subscriber is active (e.g., connected). A connected subscriber acts the same whether its subscription is durable or non-durable. The difference is in how the messaging system behaves when the subscriber is disconnected.

Proposez un schéma d'une architecture logicielle (par exemple le diagramme des classes), et une explication en quelques lignes, de la ou des structures de données envisagées, et les algorithmes, qui, au sein du patron de cette question, nous permettraient d'obtenir cette notion de souscripteur "durable".

Question2 : Patron Proxy

L'auteur d'une publication de messages doit être agréé. Pour cela un mandataire, par l'usage de la classe **ProxyPubSub,** contrôle les noms des publieurs.

La classe de tests unitaires, un extrait

```
public class TestsProxy extends junit.framework.TestCase{


  public void testProxyPubSub(){
    List<String> users = new ArrayList<String>();
    users.add("p1");users.add("p2");users.add("p3");

    PubSubI pubsub = new ProxyPubSub("pubsub",users);
    SubscriberI a = new SimpleSubscriber(pubsub, "a");
    SubscriberI b = new SimpleSubscriber(pubsub, "b");
    SubscriberI c = new SimpleSubscriber(pubsub, "c");

    Message msg = new Message("p1","test");
```

```
    assertEquals(3,pubsub.publish(new String[]{"a","c","b"}, msg));

    msg = new Message("x","test");
    assertEquals(0,pubsub.publish(new String[]{"a","c","b"}, msg));
  }
```

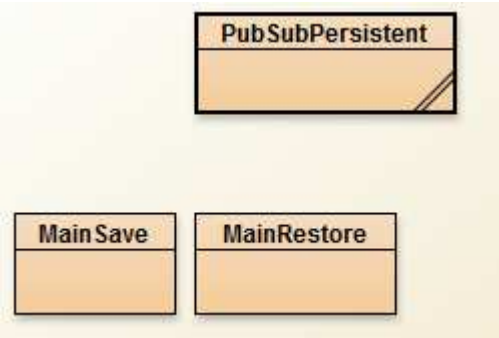
Question2)

Rappelez le principe du patron Proxy, proposez une implémentation de la classe **ProxyPubSub**. La liste des utilisateurs agréés est transmise en paramètre à la création du mandataire, sur l'exemple ci-dessus seuls p1, p2 et p3 sont habilités à publier.

Question3 : Persistance

La persistance est assurée par la classe ***PubSubPersistent***, seuls les souscripteurs inscrits sont sauvegardés, cette classe hérite de la classe ***PubSubImpl*** de la question 1.

Les classes en syntaxe UML/BlueJ:



Deux applications Java démontrant la persistance des souscripteurs.

```
public class MainSave{ // 1) effectue une sauvegarde

  public static void main(String[] args){
    PubSubI pubsub = new PubSubPersistent("pubsub");
    SimpleSubscriber a = new SimpleSubscriber(pubsub, "a");
    SimpleSubscriber b = new SimpleSubscriber(pubsub, "b");
    SimpleSubscriber c = new SimpleSubscriber(pubsub, "c");
    Message msg = new Message("p1","test");
    pubsub.publish(new String[]{"a","b"}, msg);
  }
}

public class MainRestore{ // 2) effectue une restitution

  public static void main(String[] args){
    PubSubI pubsub = new PubSubPersistent("pubsub");
      // les souscripteurs a, b et c ont été restitués
    Message msg = new Message("p1","test");
    pubsub.publish(new String[]{"a","b"}, msg); // a et b reçoivent le message
    msg = new Message("p1","test2");
    pubsub.publish(new String[]{"a","b","c","d"}, msg);
     // a, b et c reçoivent le message
  }
}
```

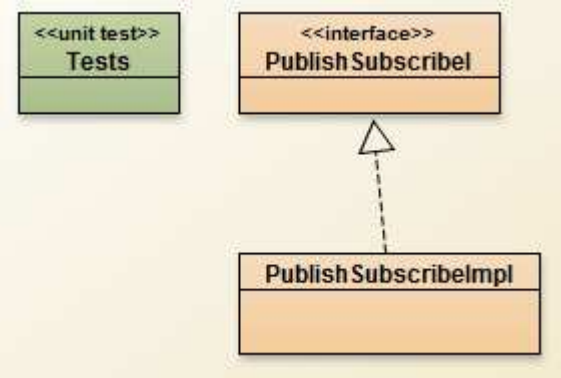
Question3)

Ecrivez une implémentation de la classe **PubSubPersistent**.

Question4: Le patron Publish/Subscribe

Ce patron permet aux abonnés ayant souscrit à un type d'événements, d'être prévenus lorsque ceux-ci se produisent. Un médiateur (classe ***PublishSubscribeImpl***) se charge de l'inscription des abonnés aux différents thèmes de souscription et assure la notification aux abonnés concernés.

Ci-dessous, en syntaxe UML/BlueJ, les classes Java :



L'interface ***PublishSubscribeI*** et la classe de tests unitaires ***Tests***

```
public interface PublishSubscribeI extends Iterable<String>{
  /** Publication pour sur thème (topic) d'un message.
   * @param topic le thème
   * @param message le message à transmettre
   * @return le nombre de souscripteurs ayant été notifiés
   */
  public int publish(String topic, Message message);

  /** Souscription à un thème (topic) de publication.
   * @param topic le thème
   * @param subscriber le souscripteur
   * @return false le souscripteur est déjà inscrit à ce thème
   */
  public boolean subscribe(String topic, SubscriberI subscriber);

  /** Obtention d'un itérateur.
   * @return un itérateur sur les thèmes enregistrés
   */
  public Iterator<String> iterator();

  /**Obtention de la liste des souscripteurs à un thème.
   * @param topic le thème
   * @return la liste des souscripteurs, une liste vide si ce thème n'existe pas
   */
  public List<SubscriberI> getSubscribers(String topic);
}
```

La classe de tests unitaires ***Tests***

```
public class Tests extends junit.framework.TestCase{

    class SubscriberTest extends SimpleSubscriber{
      Stack<Message> messages;
      SubscriberTest(String name){
        super(name);
        this.messages = new Stack<Message>();
      }
      public void onMessage(Message message)throws Exception{
        this.messages.push(message);
        if("error".equals(message.getContent()))throw new Exception();
        super.onMessage(message);
```

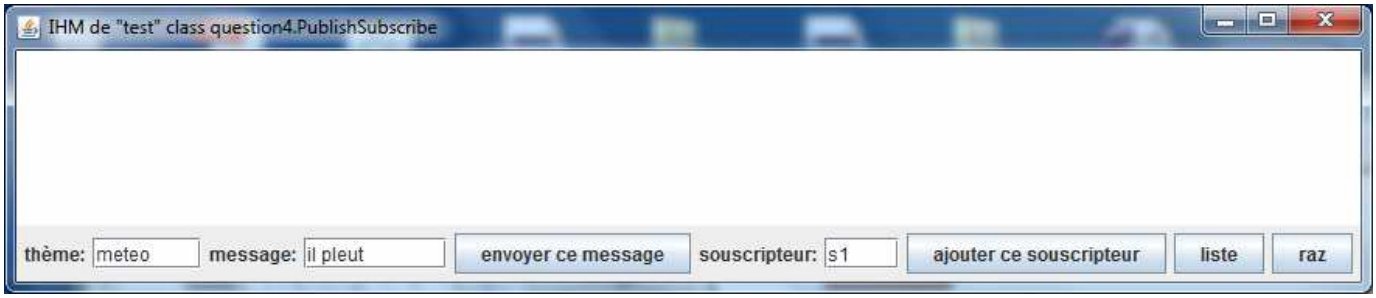
```
  }
}

public void testSimple(){
   PublishSubscribeI pubsub = new PublishSubscribeImpl();
   SubscriberTest a = new SubscriberTest("a");
   assertTrue(pubsub.subscribe("meteo",a));
   SubscriberTest b = new SubscriberTest("b");
   assertTrue(pubsub.subscribe("meteo",b));
   SubscriberTest c = new SubscriberTest("c");
   assertTrue(pubsub.subscribe("meteo",c));

   Message message = new Message("p1","il neige");

   assertEquals(3,pubsub.publish("meteo", message));
   assertEquals("il neige",a.messages.pop().getContent());
   assertEquals("il neige",b.messages.pop().getContent());
   assertEquals("meteo",pubsub.iterator().next());
}

public void testDeuxThemesUnSouscripteur(){
   PublishSubscribeI pubsub = new PublishSubscribeImpl();
   SubscriberTest a = new SubscriberTest("a");
   assertTrue(pubsub.subscribe("meteo",a));
   assertTrue(pubsub.subscribe("verglas",a));
   Message message = new Message("p1","il gèle");
   assertEquals(1,pubsub.publish("meteo", message));
   assertEquals(1,pubsub.publish("verglas", message));
}
```

Question4)
Proposez une implémentation de la classe ***PublishSubscribeImpl.***

Nota bene : il est conseillé d'utiliser les classes de la question1 afin de répondre à cette question.

Question5: Une interface graphique

Une interface graphique en Swing a été développée dans le but de "tester" une instance de PublishSubscribeImpl issue de la question 4.



Plusieurs scenarii de tests sont ainsi proposés:

Envoyer un message : L'utilisateur/testeur précise le thème et un contenu de message, au clic sur le bouton **« envoyer ce message»**, le message est publié aux souscripteurs inscrits pour ce thème.

Ajouter un souscripteur: L'utilisateur/testeur clique sur le bouton **« ajouter ce souscripteur »** alors le souscripteur est ajouté à la liste pour ce thème.

Liste : L'utilisateur/testeur obtient la liste de tous les thèmes ainsi que leurs souscripteurs.

Remise à zéro (raz) : Supprime tous les souscripteurs quelque soit le thème de souscription.

Ci-dessous, un exemple d'affichage, après l'ajout de 3 souscripteurs "**s1**","**s2**","**s3**" avec le thème **meteo** et l'envoi du message "**il pleut**"

Question5)

Complétez l'implémentation de la classe **IHM**.

```
import question1.*;
import question4.*;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import java.util.*;
import java.io.*;

public class IHM extends JFrame {

  private JTextArea         resultat = new JTextArea("", 7,60);
  private JButton           creerTheme = new JButton("créer ce thème");
  private JTextField        theme = new JTextField("meteo",6);
  private JTextField        message = new JTextField("il pleut",8);
  private JButton           envoyerMessage = new JButton("envoyer ce message");
  private JTextField        souscripteur = new JTextField("s1",4);
  private JButton           ajouterSouscripteur = new JButton("ajouter ce souscripteur");
  private JButton           liste = new JButton("liste");
  private JButton           raz = new JButton("raz");
  private PublishSubscribeI pubsub;

  public class SubscriberIHM extends SimpleSubscriber{
    public SubscriberIHM(String name){
      super(name);
    }

    @Override
    public void onMessage(Message message){
      resultat.setText(resultat.getText() + "\n" + message + "\n");
    }
  }

  public IHM() {
    this.setTitle("IHM de \"test\" class question4.PublishSubscribe");
    this.pubsub = new PublishSubscribeImpl();
    Container container = this.getContentPane();
    container.setLayout(new BorderLayout());
    container.add(resultat, BorderLayout.NORTH);
    JPanel panel = new JPanel();
    panel.setLayout(new FlowLayout());
    panel.add(new JLabel("thème:"));
    panel.add(theme);
    //panel.add(creerTheme);
    panel.add(new JLabel("message:"));
```

```
    panel.add(message);
    panel.add(envoyerMessage);
    panel.add(new JLabel("souscripteur:"));
    panel.add(souscripteur);
    panel.add(ajouterSouscripteur);
    panel.add(liste);
    panel.add(raz);
    container.add(panel, BorderLayout.SOUTH);

     this.ajouterSouscripteur.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent ae) {

// à compléter, indiquez sur votre copie "code pour ajouter"

           IHM.this.pack();
         }

       });

       this.envoyerMessage.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent ae) {

// à compléter, indiquez sur votre copie "code pour envoyer"

           IHM.this.pack();
         }
       });

     this.liste.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent ae) {

// à compléter, indiquez sur votre copie "code pour liste"

           IHM.this.pack();
         }
       });

     this.raz.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent ae) {


// à compléter, indiquez sur votre copie "code pour raz"



           IHM.this.pack();
         }
       });

     this.pack();
     this.setVisible(true);
   }


   public static void main() {
     new IHM();
   }
```

Nota bene : Une idée de la solution sera en ligne à cette URL : http://jfod.cnam.fr/NFP121/annales/2017_janvier/

Annexes

java.util
Interface Collection<E>

```
public interface Collection<E>
extends Iterable<E>
```

Method Summary	
boolean	**add**(E e) Ensures that this collection contains the specified element (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear**() Removes all of the elements from this collection (optional operation).
boolean	**contains**(Object o) Returns `true` if this collection contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns `true` if this collection contains all of the elements in the specified collection.
boolean	**equals**(Object o) Compares the specified object with this collection for equality.
int	**hashCode**() Returns the hash code value for this collection.
boolean	**isEmpty**() Returns `true` if this collection contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this collection.
boolean	**remove**(Object o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this collection.
Object[]	**toArray**() Returns an array containing all of the elements in this collection.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

java.util
interface List<E> ***partielle***

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Implementing Classes:

LinkedList, … ArrayList, Vector

---

public interface **List<E>** extends Collection<E>

Method Summary	
boolean	**add**(E e) Appends the specified element to the end of this list (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns true if this list contains the specified element.
boolean	**isEmpty**() Returns true if this list contains no elements.
E	**get**(int index) Returns the element at the specified position in this list.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
int	**size**() Returns the number of elements in this list.

---

java.util.
Interface Map<K,V> ***partielle***

See Also:

HashMap, TreeMap, Hashtable

Nested Class Summary	
static interface	**Map.Entry<K, V>** A map entry (key-value pair).

Method Summary	
void	**clear**() Removes all mappings from this map (optional operation).
boolean	**containsKey**(Object key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K, V>>	**entrySet**() Returns a set view of the mappings contained in this map.
boolean	**equals**(Object o) Compares the specified object with this map for equality.
V	**get**(Object key) Returns the value to which this map maps the specified key.
int	**hashCode**() Returns the hash code value for this map.

boolean	**isEmpty**() Returns `true` if this map contains no key-value mappings.
Set<K>	**keySet**() Returns a set view of the keys contained in this map.
V	**put**(K key, V value) Associates the specified value with the specified key in this map (optional operation).
void	**putAll**(Map<? extends K,? extends V> t) Copies all of the mappings from the specified map to this map (optional operation).
V	**remove**(Object key) Removes the mapping for this key from this map if it is present (optional operation).
int	**size**() Returns the number of key-value mappings in this map.
Collection<V>	**values**() Returns a collection view of the values contained in this map.

java.io
Class ObjectOutputStream

Constructor Summary	
	ObjectOutputStream(OutputStream out) Creates an ObjectOutputStream that writes to the specified OutputStream.

Method Summary	
void	**writeObject**(Object obj) Write the specified object to the ObjectOutputStream.

java.io
Class ObjectInputStream

Constructor Summary	
	ObjectInputStream(InputStream out)

Method Summary	
Object	**readObject**(Object obj)

```
public abstract class Subscriber implements SubscriberI{




  public Subscriber(String name){







  }
  public String getName(){


    return
  }

  public List<PubSubI> getPubSubList(){



    return
  }

  public boolean addPubSub(PubSubI pubsub){









    return false;
  }


//...
}
```

```
public class PubSubImpl implements PubSubI{



  public PubSubImpl(String name){



  }

  public String getName(){

    return
}

  public boolean publish(String name, Message message){









    return
  }

  public int publish(String[] names, Message message){
    int number = 0;







    return number;
  }

  public boolean subscribe(SubscriberI subscriber){







    return
  }

  public Iterator<SubscriberI> iterator(){



    return
}


}
```

```
// Question 4
public class PublishSubscribeImpl implements PublishSubscribeI{




  public PublishSubscribeImpl(){




  }

  public int publish(String topic, Message message){
    int res = 0;









    return res;
  }

  public boolean subscribe(String topic, SubscriberI subscriber){









    return
}

  public Iterator<String> iterator(){




    return

}

  public List<SubscriberI> getSubscribers(String topic){




}
```