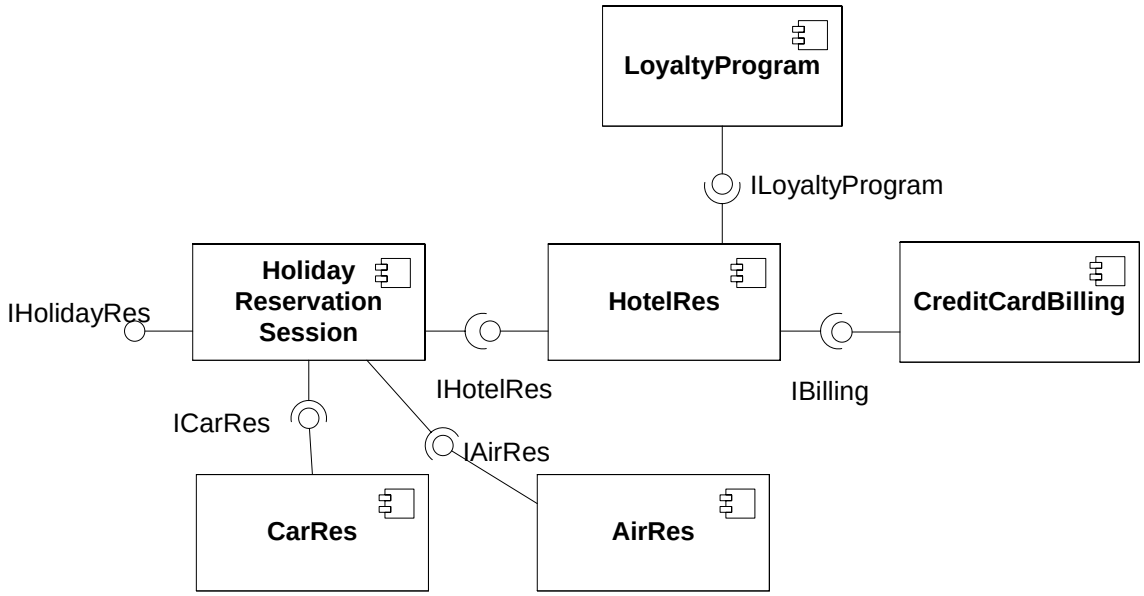
Tutorial 1 – Simple Architectures in UML2

Exercise 1 – Hotel Reservation System

Recall the expedia-style hotel reservation system discussed in lectures.

- `HolidayReservationSession` is a component that provides all the functionality required to book a holiday. It exposes this functionality via the provided interface `IHolidayRes`. It requires three components that implement the interfaces `ICarRes`, `IAirRes` and `IHotelRes`.
- `HotelRes` is a component that provides the functionality for booking a hotel from a selection of possible hotels (of the same hotel chain). It implements the `IHotelRes` interface, consisting of operations relating to hotel-booking. It requires two components that implement the `ILoyaltyProgram` and `IBilling`.
- `CreditCarBilling` is a component that implements the `IBilling` interface, providing operations related to charging a credit card.
- `LoyaltyProgram` is a component that implements the `ILoyaltyProgram` interface, providing operations related to a loyalty program for guests use the hotel chain
- `CarRes` implements `ICarRes`, providing functionality related to booking a car
- `AirRes` implements `IAirRes`, provding functionality related to booking an airplane.

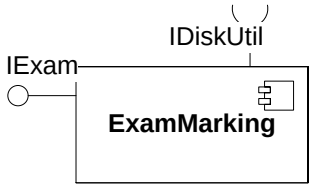
1. Define *detailed interface descriptions* in UML 2 for the `HolidayReservationSession` component, the `HotelRes` component and the `LoyaltyProgram` component! Make the interfaces as realistic as possible! You can use the space on the next page for providing your definitions.



2. Write Java code to implement the `HolidayReservationSession` component using the mapping from the lecture. Note that, in the lecture, we have already written code for the constructor and the instantiation of the components. In this exercise, you are asked to provide an implementation of one of the methods from `IHolidayRes`. You can use the space below to write your code.

Exercise 2 – Exam Marking Component

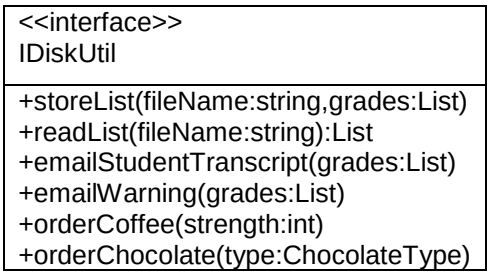
Consider the following specification of an exam marking component, `ExamMarking`. This component is to be used within the Department of Informatics' new student administration system (not depicted here). Users of the system will be academics, administrators and students.
The component exposes one provided interface, `IExam`, defining functionality for entering marks for students, performing standardization, and viewing marks in various kinds of presentation.
The component also has one required interface, `IDiskUtil`, defining functions relating to storing marks on disk and emailing marks.
Here is how the component is represented in UML 2 – with an interface specification included.



<<interface>> IExam
+InputGrade(ID:int,mark:float,code:SubjectCode) +CalculateOverallAverage():float +CalculateOverallDistribution(code:SubjectCode):Distribution +Standardize():void +ViewMark(ID:int,code:SubjectCode):float +CalculateStudentAverage(ID : int):float +FixNiceSnack()

`IExam` interface semantics:

- `InputGrade` takes in the final grade for a student for a subject. `ID` is the student's ID, `mark` is the student's grade and `code` is the code of the subject being graded.
- `CalculateOverallAverage` computes the overall average grade for all students in all subjects – this is returned as a floating point.
- `CalculateDistribution` computes the overall mark distribution for a subject – this is returned as an element of the `Distribution` data structure. `code` is the code of the subject.
- `Standardize` is a function that standardizes mark distributions for all subjects.
- `ViewMark` retrieves the grade of a student – this is returned as a float. `ID` is the student's ID, `code` is the subject code.
- `CalculateStudentAverage` computes the average grade for a particular student, returning this as a float. `ID` is the student's ID.
- `FixNiceSnack` orders a nice snack for Steffen.



`IDiskUtil`

- `storeList` writes the list of grades to a file. `fileName` is the name of the file. `grades` is the list of grades for all students in all subjects.
- `readList` reads a list of grades for all students in all subjects from a file, returning them as an element of the data structure `List`. `fileName` is the name of the file to be read from.
- `emailStudentTranscript` emails a transcript of grades to each student mentioned in the `grades` list. `grades` is the list of grades for all students in all subjects.
- `emailWarning` emails threats of expulsion from the College to all students mentioned in the `grades` list whose average falls below a certain point. `grades` is the list of grades for all students in all subjects.
- `orderCoffee()` utilizes the Department's automated telephony system and calls Caffé Nero to order Steffen his favourite espresso. `strength` is the number of espresso shots required.
- `orderChocolate()` uses the telephony system to order Steffen a bar of chocolate. `type` is kind of chocolate required.

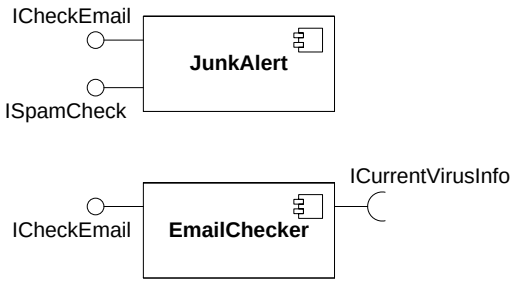
What is wrong with this component specification? Fix it, bearing in mind the discussion on what makes a component from last week's lecture. (Note there are several things that are obviously wrong with this component, and several more subtle problems.) Use the space below to write down your answers.

Exercise 3 – Component Substitution

Consider the following two components written by rival software houses. The components are meant to work as plug-ins with a GUI-based email client.

- `EmailChecker` is a component that scans incoming emails for viruses. The interface that provides this functionality is called `ICheckEmail`. The component requires the use of another component that provides an up-to-date list of current viruses. This required functionality is defined in an interface called `ICurrentVirusInfo`.
- `JunkAlert` does the same kind of thing as `EmailChecker`, by providing an alternative implementation of the `ICheckEmail` interface. Also, it has additional spam checking functionality, implemented by the `ISpamCheck` interface. It does not require another component to provide a list of viruses, as it has a magic algorithm inside it that can detect viruses and spam on the fly.

The components are drawn in UML2 as follows:



1. Can `EmailChecker` be substituted by `JunkAlert` within an application?
2. Can `JunkAlert` be substituted by `EmailChecker`?
3. Draw two example architectures to back up your reasons (i.e., one application consisting of other components interacting with `EmailChecker` and the same application but with `JunkAlert` replacing `EmailChecker`)!

Exercise 4 – Reverse Engineering an Architecture *(advanced)*

At the end of this document is a rather intimidating looking Java program. The purpose of the code is to send emails from the President of Marcuul to a list of people, alerting potential financial partners of a great business opportunity.

The President has heard that software architecture aids maintenance—unfortunately, his current team of hackers didn't learn about architecture when they studied at the University of South-East Marcuul. He has therefore employed you as a consultant brought in to help with documenting the architecture of this code.

This is a common consulting role: to work backwards from code to design, and then (possibly) to aid in making improvements.

Look at the Java code and draw a *simple* components-and-connector architectural diagram in UML2.

Remember to *abstract away* parts of the code that are not architecturally relevant. In the lectures we saw how abstract components can be implemented as classes. However, always remember that not all classes correspond to components!

You don't need to understand the code in much detail. You are a senior consultant, and haven't actually written a program in years—and that was written in Smalltalk. The main thing is to focus on the *coarse grain* aspects of the code, ignoring fine grain stuff like algorithms, method content and data structures as much as you can.

The code begins on the next page. Use the space below to draw your architecture diagram.

```
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * This class contains the main function and connects all the objects
 * of classes Invitation and Transport together...
 */
public class System {
  public static void main (String args[]) {
    Transportable client = new Transport ("marcuul.gov.net", 25);
    Address[] friends = {new Address ("allstaff@dcs.kcl.ac.uk"),
                         new Address ("bill@microsoft.com"),
                         new Address ("steve@mac.com"),
                         new Address ("david@numberten.gov.uk"),
                         new Address ("pdiddy@badboy.biz")};
    Invitation invite = new Invitation (friends, client);
    invite.sendInvitation();
}

/**
 * Invitation class.
 *
 * This class handles all the business logic to do with
 * alerting interesting people in possible financial opportunities with
 * the President. It acts as a client of the Transport class via the
 * Transportable interface, using that class to connect to the email
 * server and to send messages.
 */
public class Invitation {
  // this field lists address of all the President's friends
  private Address[] listOfFriends;

  // myEmailClient stores a reference to an email client that can send
  // messages via smtp - this is object of interface type Transportable
  Transportable myEmailClient

  public Invitation (Address[] addressList, Transportable client) {
    this.listOfFriends = addressList;
    this.cmyEmailClient = client;
  }

  public sendInvitation() {
    Message msg = new Message();

    msg.setFrom (new Address ("President Fred Shady",
                              "president@marcuul.net"));
    msg.setRecipients (RecipientType.TO, listOfFriends);
    msg.setSubject ("Exciting financial proposition!!!!");
    msg.setText (
      "Most esteemed kind sir/madam, I wish to alert you personally as
       the President of Marcuul of an exciting new business
       opportunity. We are currently interested in foreigners who might
       be able to invest our budget surplus, with a guaranteed 50%
       share of profits. Click on the following link
       http://www.prettyshadyventures.com/creditcarddetails.cgi and
       enter your credit card number and expiry date as verification of
       your interest in this venture. Then my associates will deposit
       the 50,000,000 Marcuul dollars directly into your account....."
    );
    myEmailCient.send(msg);
  }
}


/**
 * Address class
 *
 * This class is designed to store an email address.
 */
public class Address {
```

```
  private String address;
  private String personal;

  public Address (String personal, String address) {
    setPersonal (personal);
    setAddress (address);
  }

  public Address (String address) {
    this (null, address);
  }

  public String getAddress() {
    return address;
  }

  public String getPersonal() {
    return personal;
  }

  public void setAddress( String address ) {
    this.address = address
  }

  public void setPersonal( String name ) {
    this.personal = name;
  }
}
```

```
/**
 * Message class
 *
 * This class is designed to build an email message. You should use
 * this class to create a message object which is passed to a Transport
 * object which sends the email.
 *
 * Here's what I would do to set up a simple Message object:
 *
 * Message msg = new Message();
 * msg.setFrom (new Address ("Bill", "bill@coolservlets.com"));
 * msg.setRecipient (RecipientType.TO,
 *                   new Address ("Matt", "matt@coolservlets.com" ) );
 * msg.setSubject ("Hello!");
 * msg.setText ("Hey Matt. Hope everything is going well....");
 *
 * To add a CC recipient, do this:
 * msg.setRecipient (RecipientType.CC,
 *                   new Address ("A Friend",
 *                                "someone@someplace.com"));
 */
public class Message {
  private Address   from;
  private Address[] recipientTO;
  private Address[] recipientCC;
  private Address[] recipientBCC;
  private Address   replyTo;
  private String    subject;
  private String    msgText;

  public Message() {
    recipientTO = new Address[0];
    recipientCC = new Address[0];
    recipientBCC = new Address[0];
  }

  /**
   * Adds one email address to the existing list of addresses.
   */
  public void addRecipient (int type, Address address) {
    Address[] current = getRecipients (type);
    Address[] newAddr = new Address[current.length + 1];
    System.arrayCopy (current, 0, newAddr, 0, current.length);
    newAddr[current.length] = address;
    setRecipients (type, newAddr);
  }

  /**
   * Sets the Reply-To address.
   */
  public void setReplyTo (Address replyToAddress) {
    replyTo = replyToAddress;
  }

  /**
   * Returns the Reply-To address.
   */
  public Address getReplyTo() {
    return replyTo;
  }

  /**
   * Returns the specified recipient array.
   */
  public Address[] getRecipients (int type) {
    if (type == RecipientType.TO) {
      return this.recipientTO;
    }
    else if (type == RecipientType.CC) {
      return this.recipientCC;
    }
```

```
  else { //if (type == RecipientType.BCC)
    return this.recipientBCC;
  }
}

/**
 * Returns the FROM address in an array of addresses.
 * If no FROM address has been specified, the method returns null.
 */
public Address[] getFrom() {
  if (from != null) {
    Address[] tempAddr = new Address[1];
    tempAddr[0] = from;
    return tempAddr;
  } else {
    return null;
  }
}

/**
 * Returns the message subject.
 */
public String getSubject() {
  return subject;
}

/**
  * Returns the message text.
  */
public String getText() {
  return msgText;
}

/**
  * Sets the FROM email address. (This will replace the current FROM
  * address since there can be only one FROM address.)
  */
public void setFrom (Address address) {
  from = address;
}

/**
  * This method will set one recipient and will override anything
  * that has been previously stored.
  */
public void setRecipient (int type, Address address) {
  Address[] tempAddr = new Address[1];
  tempAddr[0] = address;
  setRecipients (type, tempAddr);
}

/**
  * This method will sets multiple recipients and will override
  * anything that has been previously stored.
  */
public void setRecipients (int type, Address[] addresses) {
  if (type == RecipientType.TO) {
    this.recipientTO = addresses;
  }
  else if (type == RecipientType.CC) {
    this.recipientCC = addresses;
  }
  else if (type == RecipientType.BCC) {
    this.recipientBCC = addresses;
  }
}

/**
  * Sets the SUBJECT of the email.
  */
public void setSubject (String subject) {
```

```
    this.subject = subject;
    if (this.subject == null) {
      this.subject = "[NO SUBJECT]";
    }
  }

  /**
    * Sets the TEXT of the email.
    */
  public void setText (String msgText) {
    this.msgText = msgText;
  }
}
```

```
/**
 * RecipientType interface
 *
 * This interface is only used to collect recipient type constants.
 *
 * It is primarily used like so:
 *
 * if (type == RecipientType.TO)
 *   // do TO stuff
 * else if (type == RecipientType.CC)
 *   // do CC stuff
 *
 */
public interface RecipientType {
  public static final int TO  = 1;
  public static final int CC  = 2;
  public static final int BCC = 3;
}

/**
  * Transportable Interface
  */
public interface Transportable {
  public void send (Message msg) throws TransportException;
  public String getSmtpHost();
  public int getSmtpPort();
  public void setSmtpHost (String smtpHost);
  public void setSmtpPort (int smtpPort);
}

/**
  * Transport class
  *
  * This class is the class that actually sends an email by talking to
  * an SMTP server.
  *
  * This class implements the Transportable interface.
  *
  * Here's how this class should be used:
  *
  * Message msg = new Message();
  * // fill this up, ie, FROM, TO's, etc... (see Message class)
  * Transport tr = new Transport ("yourHost.com", 25);
  * // SMTP ports are 25 by default
  *
  * try {
  *   tr.send (msg);
  * }
  * catch (TransportExeception te) {
  *   System.err.println ("TransportException! " + te);
  * }
  */
public class Transport implements Transportable {
  private String  smtpHost;
  private String  smtpResponse;
  private int     smtpPort;

  /**
    * Construct a Transport object by specifying a host and port.
    */
  public Transport (String smtpHost, int smtpPort) {
    this.smtpHost = smtpHost;
    this.smtpPort = smtpPort;
  }

  /**
    * Returns the SMTP host (e.g., smtpServer.coolservlets.com).
    */
  public String getSmtpHost( ) {
    return this.smtpHost;
  }
```

```
/**
  * Returns the SMTP port number (usually 25).
  */
public int getSmtpPort( ) {
  return this.smtpPort;
}

/**
  * Sets the SMTP host.
  */
public void setSmtpHost( String smtpHost ) {
  this.smtpHost = smtpHost;
}

/**
  * Sets the SMTP port number.
  */
public void setSmtpPort( int smtpPort ) {
  this.smtpPort = smtpPort;
}

/**
  * Send method
  * Sends the message by talking to a SMTP port.
  */
public void send (Message msg) throws TransportException {
  PrintWriter out = null;
  BufferedReader in  = null;

  try {
    // Open a connection to the SMTP port:
    Socket s = new Socket (this.smtpHost, this.smtpPort);

    // The following line now includes a "true" in the PrintWriter
    // constructor. This makes the stream flush after every println
    // by default. This fixes a bug found by Jon Barber.
    out = new PrintWriter (s.getOutputStream(), true);

    // Create a BufferedReader so we can read back response codes
    // from the SMTP port
    in = new BufferedReader (new InputStreamReader (
                                    s.getInputStream()));

    // Check to see that we connected the SMTP port correctly by
    // reading a line from the SMTP port and checking the error
    // response code. We use StringBuffers to maximize efficiency.
    if (smtpErrorExists (in, "220")) {
      throw new TransportException ("Can't connect to: " +
                                    this.smtpHost + ". Port: " +
                                    this.smtpPort + "\t" +
                                    this.smtpResponse );
    }
  }
  catch (IOException ioe) {
    throw new TransportException ("Can't connect to: " +
                                  this.smtpHost + ". Port: " +
                                  this.smtpPort + "\t " );
  }

  sendMessage (msg, in, out);
}

private void sendMessage (Message msg, BufferedReader in,
                          PrintWriter out) throws TransportException
{
  // Say hello to the SMTP port then check for the appropriate
  // response code
  out.println ("HELO " + this.smtpHost);
  if (smtpErrorExists (in, "250")) {
    throw new TransportException ("SMTP error: HELO failed.\t" +
```

```
                                    smtpResponse);
}

// Start writing our mail data to the SMTP port starting with the
// sender of the email. Actually, to send an email via SMTP, you
// don't need to included the MAIL FROM field - this allows you to
// send emails completely anonymously. In this package, I make it a
// requirement to include the MAIL FROM field just so this email
// package isn't spammer friendly 'out of the box'.
Address[] from = msg.getFrom();
if (from == null) {
  throw new TransportException (
      "SMTP error: No FROM address specified, can't send email."
    );
}
else {
  if (from[0].getAddress() != null &&
      from[0].getAddress().length() > 0 ) {
    out.println ("MAIL FROM: <" + from[0].getAddress() + ">" );
    if (smtpErrorExists (in, "250")) {
      throw new TransportException (
          "SMTP error: adding a sender failed (FROM field).\t" +
          smtpResponse);
    }
  }
  else {
    throw new TransportException (
        "SMTP error: No FROM address specified, can't send email."
    );
  }
}

// TO, CC, BCC (all are added to the recipient list)
// SMTP error checking is done after every recipient is added.
Address[] to = msg.getRecipients (RecipientType.TO);
if (to == null || to.length == 0) {
  throw new TransportException (
      "SMTP error: no RCPT TO (a recipient) specified.");
} else {
  for (int i=0; i<to.length; i++) {
    out.println ("RCPT TO: <" + to[i].getAddress() + ">");
    if (smtpErrorExists (in, "250")) {
      throw new TransportException (
          "SMTP error: adding a TO recipient failed. Error " +
          "with this address: " + to[i].getAddress() + "\t" +
          smtpResponse);
    }
  }
}

Address[] cc = msg.getRecipients (RecipientType.CC);
if (cc != null) {
  for (int i=0; i<cc.length; i++) {
    out.println ("RCPT TO: <"+ cc[i].getAddress() +">");
    if (smtpErrorExists (in, "250")) {
      throw new TransportException (
          "SMTP error: adding a CC recipient failed. Error " +
          "with this address: " + cc[i].getAddress() + "\t" +
          smtpResponse);
    }
  }
}

Address[] bcc = msg.getRecipients (RecipientType.BCC);
if (bcc != null) {
  for (int i=0; i<bcc.length; i++) {
    out.println ("RCPT TO: <"+ bcc[i].getAddress() +">");
    if (smtpErrorExists (in, "250")) {
      throw new TransportException (
          "SMTP error: adding a BCC recipient failed. Error " +
          "with this address: " + bcc[i].getAddress() + "\t" +
```

```
          smtpResponse);
    }
  }
}

// Start of data section
out.println ("DATA");
if (smtpErrorExists (in, "354")) {
  throw new TransportException (
    "SMTP error: Writing DATA field of message failed.\t" +
    smtpResponse);
}
else {
  // From:
  if (from != null) {
    out.println ("From: " + ((from[0].getPersonal() != null)?
                             from[0].getPersonal()+" " : "") +
                 "<"+ from[0].getAddress() +">" );
  }

  // To:
  if (to != null && to.length > 0) {
    StringBuffer buf = new StringBuffer();
    buf.append( "To: " );
    for (int i=0; i<to.length; i++) {
      buf.append ((to[i].getPersonal()!=null)?
                  (to[i].getPersonal() + " <" + to[i].getAddress() + ">"):
                  (to[i].getAddress()));
      if (i < to.length-1) { // add commas between emails
        buf.append (", ");
      }
    }
    out.println (buf.toString());
  }

  // CC:
  if (cc != null && cc.length > 0) {
    StringBuffer buf = new StringBuffer();
    buf.append ("CC: ");
    for (int i=0; i<cc.length; i++) {
      buf.append ((cc[i].getPersonal()!=null)?
                  (cc[i].getPersonal() + " <" + cc[i].getAddress() + ">" ):
                  (cc[i].getAddress()));
      if( i < cc.length-1 ) // add commas between emails
        buf.append( ", " );
      }
    }
    out.println( buf.toString() );
  }

  // Reply-To: (there should only be one reply-to address)
  Address replyTo = msg.getReplyTo();
  if (replyTo != null) {
    String rt = replyTo.getAddress();
    if (rt != null && rt.length() > 0) {
      StringBuffer buf = new StringBuffer();
      buf.append ("Reply-To: ");
      buf.append ((replyTo.getPersonal()!=null)?
                  (replyTo.getPersonal() + " <" +
                   replyTo.getAddress() + ">"):
                  (replyTo.getAddress()));
      out.println( buf.toString() );
    }
  }

  // Subject:
  if (msg.getSubject() != null) {
    out.println ("Subject: " + msg.getSubject());
  }
  // Email body:
  if (msg.getText() != null) {
```

```
        out.println (msg.getText());
      }
      // End:
      // Signal that we're done with the email by printing the DATA
      // section with a dot on a line by itself.
      out.println( "." );

      // Check to see if the message was successfully queued for
      // delievery:
      if (smtpErrorExists (in, "250")) {
        throw new TransportException ("Error: Message failed to be " +
                                      "sent.\t" + smtpResponse );
      }

      // Exit the SMTP port
      out.println ("QUIT");
    } // else
    out.close();
  } // end "send" method

  /**
    * Checks for errors in the SMTP response.
    */
  private boolean smtpErrorExists (BufferedReader in,
                                   String errorCode) {
    try {
      smtpResponse = in.readLine();
      if (!smtpResponse.startsWith (errorCode)) {
        return true;
      }
      return false;
    }
    catch (IOException ioe) {
      return true;
    }
  }
}

/**
  * TransportException class
  */
public class TransportException extends Exception {

  public TransportException() {
    super();
  }

  public TransportException (String msg) {
    super (msg);
  }
}
```