asynchronous interactions
learning objectives

- learn what asynchronous interactions are
- learn about asynchronous methods in java
- learn about tcp/udp sockets and web sockets
- learn about message-oriented middleware and jms*
what is an asynchronous interaction?

no blocking of the client until the server is done

no polling by the client when a result is expected from the server

asynchronous interactions allow to achieve time decoupling

synchronous

polling

asynchronous
a session bean can implement asynchronous methods. They rely on the notion of future objects, which are also called promises. The container returns the control to the client before the method is actually invoked in background. The client can try to get the result but might be blocked if it is not ready yet.
asynchronous methods

@Remote
public interface PortfolioRemote {
    public Future<Double> computeValue();
}

@Stateful
public class Portfolio implements PortfolioRemote {
    @Resource
    SessionContext context;
    ...
    @Asynchronous
    public Future<Double> computeValue() {
        double value = ...; // Processor-intensive computation
        return new AsyncResult<Double>(value);
    }
}

an asynchronous method must return
void or a Future<V> object

if it returns void, it cannot declare exceptions

the client can use the Future<V> object to retrieve
the actual result or to cancel the invocation
asynchronous methods

```java
@Remote
public interface PortfolioRemote {
    public Future<Double> computeValue();
}
```

Client side:

```java
Future<Double> value = myPortfolio.computeValue();
System.out.println("Portfolio is worth $" + value.get());
```

```java
try {
    System.out.println("Portfolio is worth $" + value.get(5, TimeUnit.SECONDS));
} catch (TimeoutException ex) {
    value.cancel(true);
    System.err.println("Timeout: operation was cancelled");
}
```

Server side:

```java
@Asynchronous
public Future<Double> computeValue() {
    if (context.wasCancelCalled()) {
        System.err.println("Call to computeValue() was cancelled");
        return null;
    }
    double value = ...;  // Processor-intensive computation
    return new AsyncResult<Double>(value);
}
```
asynchronous messaging using sockets

The OSI* model

- Application
- Presentation
- Session
- Transport
- Network
- Data link
- Physical link

transmission control protocol
internet protocol

- Stream oriented
- Reliable channels
- FIFO ordering

FIFO = first in first out

Packet oriented
Best-effort routing
Error detection
Datagram fragmentation
### Internet Protocol

An IP address is used by the IP protocol to address computers and routers.

An IP v4 address consists of 32-bits (4 bytes) and is often written in **dotted decimal format**, e.g., 130.223.171.8.

<table>
<thead>
<tr>
<th>Class</th>
<th>First byte</th>
<th>Networks</th>
<th>Hosts</th>
<th>Address format</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1→126</td>
<td>$2^7 - 2$ = 126</td>
<td>$2^{24} - 2 =$ 16'777'214</td>
<td>net id</td>
</tr>
<tr>
<td>B</td>
<td>128→191</td>
<td>$2^{14}$ = 16'384</td>
<td>$2^{16} - 2 =$ 65'534</td>
<td>net id</td>
</tr>
<tr>
<td>C</td>
<td>192→223</td>
<td>$2^{21}$ = 2'097'152</td>
<td>$2^8 - 2 =$ 254</td>
<td>net id</td>
</tr>
<tr>
<td>D</td>
<td>224→239</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>240→247</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
internet protocol

ip v4 address

<table>
<thead>
<tr>
<th>Class</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0NNNNNNNN.NNNNNNNNNN.HHHHHHHHH.HHHHHHHHH</td>
</tr>
<tr>
<td>B</td>
<td>10NNNNNN.NNNNNNNNNN.HHHHHHHHH.HHHHHHHHH</td>
</tr>
<tr>
<td>C</td>
<td>110NNNNN.NNNNNNNNNN.NNNNNNNNN.HHHHHHHHH</td>
</tr>
<tr>
<td>D</td>
<td>1110MMMM.MMMMMMMM.MMMMMMMM.MMMMMMMM</td>
</tr>
<tr>
<td>E</td>
<td>1111RRRR.RRRRRRRR.RRRRRRRRR.RRRRRRRRR</td>
</tr>
</tbody>
</table>

N network ID bits  M multicast address bit
H host ID bits      R reserved bits
asynchronous messaging using sockets

internet protocol

ip v6 address

addresses encoded on 128 bits

\[ 2^{128} > 3.4 \times 10^{38} \]

available addresses

\[ 2.25 \times 10^{26} \text{ km} \]

\[ \approx 10^{18} \times \text{distance earth-sun} \]

\[ 3.76 \times 10^6 \text{ km} \]

\[ \approx 10 \times \text{earth-moon} \]
asynchronous messaging
using sockets

addressing applications

an IP address designates a machine

a port number designates an application within a machine

this is known as port multiplexing

operating system

network
asynchronous messaging using sockets

Sockets are programming abstractions representing bidirectional communication channels between processes.

There exist two types of sockets, namely TCP sockets and UDP sockets.

In Java, sockets are instances of various classes found in the `java.net` package.
Asynchronous messaging using sockets

Transmission control protocol

TCP and UDP exhibit dual features

User datagram protocol

<table>
<thead>
<tr>
<th></th>
<th>Connection Oriented</th>
<th>Reliable Channels</th>
<th>FIFO Ordering</th>
<th>Message Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>UDP</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
since tcp is connection-oriented, we have two classes for tcp sockets in java

```
public class Socket {
    public Socket(String host, int port) {...}
    public OutputStream getOutputStream() {...}
    public InputStream getInputStream() {...}
    public void close() {...}
}
```

```
public class ServerSocket {
    public ServerSocket(int port) {...}
    public Socket accept() {...}
}
```

this captures the asymmetry when establishing a communication channel
public class DictionaryServer {
    private static Map dico = Map.of("inheritance", "héritage", "distributed", "réparti");

    public static void main(String[] args) {
        ServerSocket connectionServer = null;
        Socket clientSession = null;
        PrintWriter out = null;
        BufferedReader in = null;
        try {
            connectionServer = new ServerSocket(4444);
            clientSession = connectionServer.accept();
            out = new PrintWriter(clientSession.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(clientSession.getInputStream()));
            String word, mot;
            while ((word = in.readLine()) != null) {
                mot = (String) dico.get(word);
                if (mot == null) {
                    mot = "sorry, no translation available for " + word + "!";
                }
                out.println(mot);
            }
        } catch (IOException e) {
            System.out.println(e);
            System.exit(1);
        }
    }
}
public class DictionaryClient {
    public static void main(String[] args) {
        Socket mySession = null;
        PrintWriter out = null;
        BufferedReader in = null;
        BufferedReader stdIn = null;
        try {
            if (args.length < 1) {
                System.out.println("Hostname missing.");
                System.exit(1);
            }
            mySession = new Socket(args[0], 4444);
            out = new PrintWriter(mySession.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(mySession.getInputStream()));
            stdIn = new BufferedReader(new InputStreamReader(System.in));
            String fromServer, fromUser;
            System.out.println("Go on, ask the dictionary server!");
            while (!(fromUser = stdIn.readLine()).equals("quit")) {
                out.println(fromUser);
                fromServer = in.readLine();
                System.out.println("-> " + fromServer);
            }
            out.close(); in.close(); stdIn.close(); mySession.close();
        } catch (UnknownHostException e) {
            System.err.println("Host Unknown: " + args[0]);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("No connection to: " + args[0]);
            System.exit(1);
        }
    }
}
The concept of **streams** (Unix and Java)

Streams offer a unified programming abstraction for reading and writing data.

Streams can encapsulate various types of data sources, e.g., files, byte arrays in memory, sockets, etc.

Streams can encapsulate other streams to stack up processing of the data.

In Java, streams are instances of various classes found in the `java.io` package.
printer and writer classes are special streams manipulating only characters

the concept of streams (unix and java)

Socket clientSession = connectionServer.accept();
BufferedReader in = new BufferedReader(new InputStreamReader(clientSession.getInputStream()));

data source
byte stream
character stream
buffered character stream

standard operating systems-level input and output streams are also accessed via java streams (System.in & System.out)

asynchronous messaging using sockets
The concept of object streams

The network knows nothing about objects, only bytes

So how can we send an object graph across the network?

Any Java object can be encoded into a stream of bytes and recreated from it by implementing the `java.io.Serializable` interface.

This process is known as object serialization.

```java
ObjectOutputStream out = new ObjectOutputStream(sessionWithServer.getOutputStream());
out.writeObject(senderCollection);

ObjectInputStream in = new ObjectInputStream(sessionWithClient.getInputStream());
Collection receiverCollection = (Collection) in.readObject();
```
_udp sockets_

since udp is **connectionless**, we have only one class for udp sockets in java

```java
public class DatagramSocket {
    :
    public DatagramSocket() {...}
    public DatagramSocket(int port) {...}
    public void send(DatagramPacket packet) {...}
    public void receive(DatagramPacket packet) {...}
    public void close() {...}
    :
}
```

the **DatagramPacket** class is key when working with udp sockets

```java
public class DatagramPacket {
    :
    public DatagramPacket(...) {...}
    public byte[] getData() {...}
    public InetAddress getAddress() {...}
    :
}
```

it captures the **message-oriented nature of udp sockets**
public class QuoteServer {
    public static void main(String[] args) throws Exception {
        DatagramSocket socket = null;
        BufferedReader in = null;
        socket = new DatagramSocket(4445);
        in = new BufferedReader(new FileReader("one-liners.txt"));
        String quote = null;
        boolean moreQuotes = true;
        while (moreQuotes) {
            byte[] buf = new byte[256];
            DatagramPacket packet = new DatagramPacket(buf, buf.length);
            socket.receive(packet);
            quote = in.readLine();
            if (quote == null) {
                moreQuotes = false;
            } else {
                buf = (("No more, bye!").getBytes());
            }
            InetAddress address = packet.getAddress();
            int port = packet.getPort();
            packet = new DatagramPacket(buf, buf.length, address, port);
            socket.send(packet);
        }
        socket.close();
    }
}
public class QuoteClient {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) { System.out.println("Missing hostname"); System.exit(1); }
        DatagramSocket socket = new DatagramSocket();
        InetAddress address = InetAddress.getByName(args[0]);
        BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Go on, ask for a quote by typing return!");
        while (!stdIn.readLine().equals("quit") ) {
            byte[] buf = new byte[256];
            DatagramPacket packet = new DatagramPacket(buf, buf.length, address, 4445);
            socket.send(packet);
            packet = new DatagramPacket(buf, buf.length);
            socket.receive(packet);
            String received = new String(packet.getData()).trim();
            System.out.println("-> " + received);
        }
        socket.close();
    }
}
asynchronous messaging using sockets

one-to-one communication

one-to-many communication

udp multicast

a multicast address lies in range [224.0.0.0, 239.255.255.255] and defines a multicast group

in java, udp multicast is accessed via MulticastSocket, a subclass of DatagramSocket
udp multicast

```java
public class MulticastQuoteSender {
    public static void main(String[] args) throws Exception {
        MulticastSocket socket = null;
        BufferedReader in = null;
        socket = new MulticastSocket();
        InetSocketAddress group = new InetSocketAddress(InetAddress.getByName("228.0.0.4"), 9000);
        NetworkInterface networkInterface = NetworkInterface.getByName("en0");
        socket.setTimeToLive(1);
        in = new BufferedReader(new FileReader("one-liners.txt"));
        String quote = null;
        boolean moreQuotes = true;
        while (moreQuotes) {
            Thread.currentThread().sleep(2000);
            byte[] buf = new byte[256];
            quote = in.readLine();
            if (quote == null) {
                moreQuotes = false;
                buf = ("No more, bye!").getBytes();
            } else {
                buf = quote.getBytes();
            }
            System.out.println("About to multicast: " + new String(buf));
            DatagramPacket packet = new DatagramPacket(buf, buf.length, group);
            socket.send(packet);
        }
        socket.close();
    }
}
```
public class MulticastQuoteReceiver {

    public static void main(String[] args) throws Exception {
        try (MulticastSocket socket = new MulticastSocket(9000)) {
            InetSocketAddress group = new InetSocketAddress(InetAddress.getByName("228.0.0.4"), 9000);
            NetworkInterface netInterface = NetworkInterface.getByName("en0");
            socket.joinGroup(group, netInterface);
            while (true) {
                byte[] buf = new byte[256];
                DatagramPacket packet = new DatagramPacket(buf, buf.length);
                System.out.print("Waiting for the next quote: ");
                socket.receive(packet);
                String received = new String(packet.getData());
                System.out.println(received.trim());
                if (received.contains("bye")) {
                    break;
                }
            }
            socket.leaveGroup(group, netInterface);
            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
asynchronous messaging using sockets

udp multicast

Creating a multicast route:
```
wallace-palace:~ garbi$ sudo route -nv add -net 228.0.0.4 -interface en0
```

Deleting the multicast route:
```
wallace-palace:~ garbi$ sudo route -v delete -inet 228.0.0.4
```
asynchronous messaging using web sockets

unlike http, which is a request-response protocol, the web socket protocol is message-oriented and offers full-duplex channels.

web sockets are similar to tcp sockets but they offer streams of messages, rather than streams of bytes.

most web browsers and servers support web sockets via two uri schemes:

- ws://host:port/... for unencrypted streams
- wss://host:port/... for encrypted streams

the web socket protocol is based on tcp and totally independent from http, except for the handshake phase, which done via an http request interpreted by the server as an upgrade request.

*uniform resource identifier
asynchronous messaging using web sockets

Collaborative Whiteboard App

Color: Red, Blue, Orange, Green
Shape: Square, Circle

Collaborative Whiteboard App

Color: Red, Blue, Orange, Green
Shape: Square, Circle
asynchronous messaging using web sockets

```html
<!DOCTYPE html>
<html>
<head>
<title>Start Page</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
<h1>Collaborative Whiteboard App</h1>
<table>
<tr>
<td>
<canvas id="myCanvas" width="150" height="150" style="border:1px solid #000000;"></canvas>
</td>
<td>
<form name="inputForm">
<table>
<tr>
<th>Color</th>
<td><input type="radio" name="color" value="#FF0000" checked="true">Red</td>
<td><input type="radio" name="color" value="#0000FF">Blue</td>
<td><input type="radio" name="color" value="#FF9900">Orange</td>
<td><input type="radio" name="color" value="#33CC33">Green</td>
</tr>
<tr>
<th>Shape</th>
<td><input type="radio" name="shape" value="square" checked="true">Square</td>
<td><input type="radio" name="shape" value="circle">Circle</td>
</td>
</tr>
</table>
</form>
</td>
</tr>
<script type="text/javascript" src="websocket.js"></script>
<script type="text/javascript" src="whiteboard.js"></script>
</table>
</body>
</html>
```
asynchronous messaging using web sockets

```javascript
var wsUri = "ws://" + document.location.host + document.location.pathname + "whiteboardendpoint";
console.log("wsURI: " + wsUri);

var websocket = new WebSocket(wsUri);

function sendText(json) {
  console.log("sending text: " + json);
  websocket.send(json);
}

websocket.onerror = function (evt) {
  onError(evt);
}

websocket.onmessage = function (evt) {
  onMessage(evt);
}

function onError(evt) {
  writeToScreen('<span style="color: red;">ERROR:</span> ' + evt.data);
}

function onMessage(evt) {
  console.log("received: " + evt.data);
  drawImageText(evt.data);
}
```

web socket in javascript
async messaging using web sockets

```javascript
function defineImage(evt) {
    var currentPos = getCurrentPos(evt);
    for (i = 0; i < document.inputForm.color.length; i++) {
        if (document.inputForm.color[i].checked) {
            var color = document.inputForm.color[i];
            break;
        }
    }
    for (i = 0; i < document.inputForm.shape.length; i++) {
        if (document.inputForm.shape[i].checked) {
            var shape = document.inputForm.shape[i];
            break;
        }
    }
    var json = JSON.stringify({
        "shape": shape.value,
        "color": color.value,
        "coords": {
            "x": currentPos.x,
            "y": currentPos.y
        }
    });
    drawImageText(json);
    sendText(json);
}

function drawImageText(image) {
    console.log("drawImageText");
    var json = JSON.parse(image);
    var context = canvas.getContext("2d");
    context.fillStyle = json.color;
    switch (json.shape) {
        case "circle":
            context.beginPath();
            context.arc(json.coords.x, json.coords.y, 5, 0, 2 * Math.PI, false);
            context.fill();
            break;
        case "square":
            context.fillRect(json.coords.x, json.coords.y, 10, 10);
            break;
        default:
            context.fillRect(json.coords.x, json.coords.y, 10, 10);
            break;
    }
}

var canvas = document.getElementById("myCanvas");
var context = canvas.getContext("2d");
canvas.addEventListener("click", defineImage, false);
```

1. whiteboard.js
2. [Image 12x-8 to 1920x1009]
3. [Image 1585x933 to 1799x1065]
```java
@ServerEndpoint(value = "/whiteboardendpoint", encoders = {FigureEncoder.class}, decoders = {FigureDecoder.class})
public class MyWhiteboard {

    private static Set<Session> peers = Collections.synchronizedSet(new HashSet<Session>();

    @OnMessage
    public void broadcastFigure(Figure figure, Session session) throws IOException, EncodeException {
        System.out.println("broadcastFigure: " + figure);
        for (Session peer : peers) {
            if (!peer.equals(session)) {
                peer.getBasicRemote().sendObject(figure);
            }
        }
    }

    @OnOpen
    public void onOpen(Session peer) {
        peers.add(peer);
    }

    @OnClose
    public void onClose(Session peer) {
        peers.remove(peer);
    }
}
```

```java
public class Figure {
    private JsonObject json;

    public Figure(JsonObject json) {
        this.json = json;
    }

    public JsonObject getJson() {
        return json;
    }

    public void setJson(JsonObject json) {
        this.json = json;
    }

    @Override
    public String toString() {
        StringWriter writer = new StringWriter();
        Json.createWriter(writer).write(json);
        return writer.toString();
    }
}
```
public class FigureDecoder implements Decoder.Text<Figure> {

    @Override
    public Figure decode(String s) throws DecodeException {
        JsonObject jsonObject = Json.createReader(new StringReader(s)).readObject();
        return new Figure(jsonObject);
    }

    @Override
    public boolean willDecode(String s) {
        try {
            Json.createReader(new StringReader(s)).readObject();
            return true;
        } catch (JsonException ex) {
            ex.printStackTrace();
            return false;
        }
    }

    @Override
    public void init(EndpointConfig config) {
        System.out.println("init");
    }

    @Override
    public void destroy() {
        System.out.println("destroy");
    }
}

public class FigureEncoder implements Encoder.Text<Figure> {

    @Override
    public String encode(Figure figure) throws EncodeException {
        return figure.getJson().toString();
    }

    @Override
    public void init(EndpointConfig config) {
        System.out.println("init");
    }

    @Override
    public void destroy() {
        System.out.println("destroy");
    }
}
Asynchronous messaging using message-oriented middleware

- In addition to time decoupling, message-oriented middleware also achieve space decoupling.
- Time decoupling $\Rightarrow$ asynchrony
- Space decoupling $\Rightarrow$ anonymity

A message-oriented middleware is a software layer acting as a kind of “middle man” between distributed clients (of the middleware).
asynchronous messaging using message-oriented middleware

messages can be exchanged between clients written in any language†

the middleware is often based on a centralized server and a client library

† assuming a library exists for that language

many software providers offer such middleware products, e.g., IBM, Oracle, Apache
asynchronous messaging using message-oriented middleware

point-to-point model

one-to-one communication where producers send messages and each message is consumed by one and only one consumer

publish/subscribe model

one-to-many communication where producers publish messages and all consumers that have subscribed receive them

- messages are kept in a queue until consumed
- this model can be used for load-balancing but then the fifo* ordering of is no longer guaranteed

* fifo order = first in first out

- there exists various message routing criteria
  - topic-based routing
  - content-based routing
  - location-based routing
  - ...

client library
producer
message queue
consumer

client library
message routing
caller
subscriber
asynchronous messaging using message-oriented middleware

The Java Messaging Service (JMS) defines the standard asynchronous messaging API* of the Java EE platform.
when using jms, we can distinguish three key phases

1. development phase
2. deployment phase
3. execution phase

the jms api is essentially an interface-based specification that is encapsulating existing implementations from software providers

asynchronous messaging using message-oriented middleware

the jms api is essentially an interface-based specification that is encapsulating existing implementations from software providers

when using jms, we can distinguish three key phases

1. development phase
2. deployment phase
3. execution phase
asynchronous messaging using message-oriented middleware

**execution**

1. a **producer** creates and sends messages via the JMS API, specifying a destination
2. a JMS-compliant middleware routes those messages to the specified destination
3. a **consumer** receives messages via the JMS API specifying the same destination

**deployment**

1. **start** the JMS-compliant middleware broker
2. **create** the destination referenced by the producer and consumer code
3. **package** the library implementing the JMS API on the producer and consumer code
public class OrderProducer {

   @Resource(mappedName = "jms/DOPConnectionFactory")
   private static ConnectionFactory connectionFactory;

   @Resource(mappedName = "jms/DOPOrderQueue")
   private static Queue queue;

   public static void main(String[] args) throws IOException {
      JMSContext context = connectionFactory.createContext();
      JMSProducer producer = context.createProducer();
      boolean moreOrders = true;
      while (moreOrders) {
         String order = askForOrder(3);
         if (order != null) {
            System.out.println("Sending order to " + order);
            producer.send(queue, order);
            if (order.toLowerCase().contains("quit")) {
               moreOrders = false;
            }
         } else {
            moreOrders = false;
         }
      }
      System.out.println("Bye bye!");
   }
}

asynchronous messaging using message-oriented middleware
asynchronous messaging
using message-oriented middleware

development
point-to-point model
↓
destination = queue

non-blocking version

```java
public class OrderConsumer implements MessageListener {
  @Resource(mappedName = "jms/DOPConnectionFactory")
  private static ConnectionFactory connectionFactory;
  @Resource(mappedName = "jms/DOPOrderQueue")
  private static Queue queue;

  private static boolean stopReceiving = false;

  public static void main(String[] args) throws InterruptedException {
    JMSContext context = connectionFactory.createContext();
    JMSConsumer consumer = context.createConsumer(queue);
    System.out.println("I am now ready to receive orders");
    MessageListener listener = new OrderConsumer();
    consumer.setMessageListener(listener);
    for (int i = 0; i < 60; i++) {
      Thread.sleep(1000);
      System.out.print(".");
      if (stopReceiving) {
        break;
      }
    }
    System.out.println("\n Bye bye!");
  }

  @Override
  public void onMessage(Message message) {
    System.out.println();
    String order = "quit";
    try {
      order = ((TextMessage) message).getText();
      System.out.println("I received the order to " + order);
    } catch (JMSException ex) {
      System.err.println("Error when trying to receive message: " + ex.getMessage());
    }
    System.out.println("------");
    if (order.toLowerCase().contains("quit")) {
      stopReceiving = true;
    }
  }
}
```
asynchronous messaging
using message-oriented middleware

public class OrderConsumer {

@Resource(mappedName = "jms/DOPConnectionFactory")
private static ConnectionFactory connectionFactory;

@Resource(mappedName = "jms/DOPOrderQueue")
private static Queue queue;

public static void main(String[] args) throws InterruptedException {
    JMSContext context = connectionFactory.createContext();
    JMSConsumer consumer = context.createConsumer(queue);

    System.out.println("I am now ready to receive orders");

    while (true) {
        String order = consumer.receiveBody(String.class);
        System.out.println("The order is to " + order);
        if (order.toLowerCase().contains("quit")) {
            break;
        }
    }
    System.out.println("\nBye bye!");
}

String order = consumer.receiveBody(String.class, 1000);
asynchronous messaging using message-oriented middleware

development

publish/subscribe model

destination = topic

public class NewsPublisher {

    @Resource(mappedName = "jms/DOPConnectionFactory")
    private static ConnectionFactory connectionFactory;

    @Resource(mappedName = "jms/DOPNewsTopic")
    private static Topic topic;

    public static void main(String[] args) throws IOException, InterruptedException {
        JMSContext context = connectionFactory.createContext();
        JMSProducer producer = context.createProducer();
        int numberOfNews;

        do {
            numberOfNews = askForNumberOfNews(3);
            for (int i = 0; i < numberOfNews; i++) {
                String news = getNextNews();
                System.out.println("Sending news: " + news);
                producer.send(topic, news);
                Thread.sleep(2000);
            }
        } while (numberOfNews > 0);

        System.out.println("No more news to send. Bye bye!");
        producer.send(topic, "quit");
    }
}
asynchronous messaging
using message-oriented middleware

development

publish/subscribe model

destination = topic

```
public class NewsSubscriber implements MessageListener {
    @Resource(mappedName = "jms/DOPConnectionFactory")
    private static ConnectionFactory connectionFactory;

    @Resource(mappedName = "jms/DOPNewsTopic")
    private static Topic topic;

    private static boolean stopReceiving = false;

    public static void main(String[] args) throws InterruptedException {
        JMSContext context = connectionFactory.createContext();
        JMSConsumer consumer = context.createConsumer(topic);

        MessageListener listener = new NewsSubscriber();
        consumer.setMessageListener(listener);

        for (int i = 0; i < 60; i++) {
            Thread.sleep(1000);
            System.out.print(".");
            if (stopReceiving) {
                break;
            }
        }
        System.out.println("\nBye bye!");
    }

    @Override
    public void onMessage(Message message) {
        System.out.println();
        String news = "quit";
        try {
            news = ((TextMessage) message).getText();
            System.out.println("I received the following news: "+ news);
        } catch (JMSException ex) {
            System.err.println("Error when trying to receive message: "+ ex.getMessage());
        }
        System.out.println("------");
        if (news.toLowerCase().contains("quit")) {
            stopReceiving = true;
        }
    }
}
```
asynchronous messaging
using message-oriented middleware
development with message-driven enterprise beans

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue = "jms/DOPOrderQueue"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue")
})
public class OrderConsumer implements MessageListener {
    @Override
    public void onMessage(Message message) {
        try {
            System.out.println(this + " received the order to " + ((TextMessage) message).getText());
        } catch (JMSException ex) {
            System.err.println("Error when trying to receive message: " + ex.getMessage());
        }
    }
}
```

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue = "jms/DOPNewsTopic"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Topic")
})
public class NewsSubscriber implements MessageListener {
    @Override
    public void onMessage(Message message) {
        try {
            String news = ((TextMessage) message).getText();
            System.out.println(this + " received the following news: " + news);
        } catch (JMSException ex) {
            System.err.println("Error when trying to receive message: " + ex.getMessage());
        }
    }
}
```

point-to-point
publish/subscribe
non-blocking version
by definition
asynchronous messaging using message-oriented middleware

A JMS message is composed of three parts:
- A header holding required fields for the message broker, e.g., priority, time-to-live, etc.
- A list of optional properties acting as meta-data for the message selection mechanism.
- A body containing the actual data of the message (what that is).

A JMS message can be of various types, based on what data is in its body:
- **TextMessage** message = context.createTextMessage("Hello world!");
- **ObjectMessage** message = context.createObjectMessage(someSerializableObject);
- **MapMessage** message = context.createMapMessage();
- **Message** message = context.createMessage();

A consumer can select messages in terms of their properties (meta-data):

```
String selector = "name LIKE 'Max' OR (age > 18 OR address LIKE 'Lausanne')";
consumer = context.createConsumer(topic, selector);
```

```
Message message = session.createMessage();
message.setStringProperty("name", "Bob");
message.setIntProperty("age", 30);
message.setStringProperty("address", "Lausanne");
```
compared to other asynchronous messaging solutions, a message-oriented middleware offers flexible quality of service expressed in terms of:

- message ordering, priorities & time-to-live
- acknowledgement modes & transactions
- durable subscribers
- delivery modes
asynchronous messaging
using message-oriented middleware

message ordering

- messages are received in the order* they were sent with respect to a given producer
- no ordering is guaranteed across producers created from different contexts

message priorities

- priorities allow programmers to have finer control over ordering of messages
- priorities range from 0 (lowest) to 9 (highest), e.g., producer.setPriority(5);

message time-to-live

- the time-to-live specifies how long the broker should keep the message at most
- the time-to-live is expressed in milliseconds, e.g., producer.setTimeToLive(20000);
asynchronous messaging using message-oriented middleware

acknowledgment modes and transactions

- acknowledgment informs the broker that the client did received a message

<table>
<thead>
<tr>
<th>Acknowledgment Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTO_ACKNOWLEDGE</td>
<td>Messages are automatically acknowledged by the context.</td>
</tr>
<tr>
<td></td>
<td>```java</td>
</tr>
<tr>
<td></td>
<td>context = connectionFactory.createContext(JMSContext.AUTO_ACKNOWLEDGE);</td>
</tr>
<tr>
<td>DUPS_OK_ACKNOWLEDGE</td>
<td>Messages are automatically acknowledged by the context but duplicate messages are possible in case of failures (but more efficient than AUTO_ACKNOWLEDGE).</td>
</tr>
<tr>
<td>CLIENT_ACKNOWLEDGE</td>
<td>Client acknowledges messages itself, invoking <code>acknowledge()</code> on each message.</td>
</tr>
<tr>
<td>SESSION_TRANSACTED</td>
<td>Messages are grouped in the context of local transactions that are committed by explicitly calling <code>context.commit()</code> or rolled back by calling <code>context.rollback()</code></td>
</tr>
</tbody>
</table>

as soon as messages are sent or received via a transacted session, the first transaction starts and those messages will be grouped, until the client calls either `context.commit()` or `context.rollback()` after this call, the current transaction terminates and a new one is started.
The termination of a transaction affects a producer and a consumer as follows:

**On a producer,** what happens to messages sent during that transaction?
- After `context.commit()`, all messages are **effectively sent**.
- After `context.rollback()`, all messages are **disposed**.

**On a consumer,** what happens to messages received during that transaction?
- After `context.commit()`, all messages are **disposed**.
- After `context.rollback()`, all messages are **recovered and might be delivered again** as part of the newly started transaction.

Why “might be delivered again” and not “will be delivered again”? 
asynchronous messaging using message-oriented middleware

durable subscribers

- in the publish/subscribe model, messages are only received by subscribers that are connected at the time of the publication and lost for late comers
- a durable subscriber is one that needs to receive all messages published on a topic, even those published when it was disconnected from the broker
- to tell the broker what messages should be kept for a disconnected durable subscriber, we must provide a unique name identifying that subscriber:
  
  ```java
  consumer = context.createDurableConsumer(topic, "fomo");
  ```
- to delete the state maintained by the broker for a durable subscriber:
  
  ```java
  context.unsubscribe("fomo");
  ```
asynchronous messaging
using message-oriented middleware

Delivery modes

- delivery modes allow to balance transport reliability and throughput, depending whether the occasional loss of messages is tolerable or not.
- In JMS, there exists two delivery modes:
  - NON_PERSISTENT: most efficient but less reliable, since messages are guaranteed to be delivered at most once, i.e., some might be lost due to network failures.
    ```java
    producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
    ```
  - PERSISTENT: most reliable, since messages are guaranteed to be delivered once and only once, which is usually achieved by persisting sent messages on stable storage.
    ```java
    producer.setDeliveryMode(DeliveryMode.PERSISTENT);
    ```