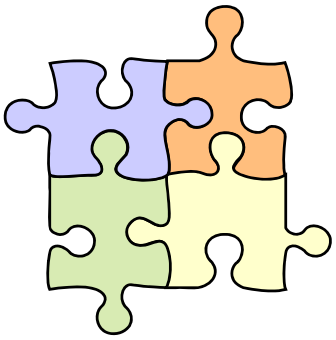
Luca Cabibbo

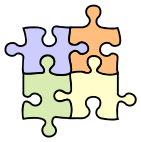


Architetture Software

Messaging

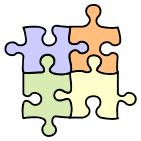
Dispensa MW 4

ottobre 2008



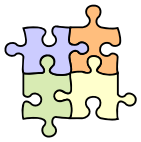
- Fonti

- The Java EE 5 Tutorial – The Java Message Service API – *http://java.sun.com/javaee/5/docs/tutorial/doc/*



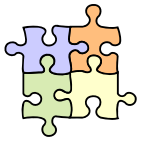
- Obiettivi e argomenti

- Obiettivi
  - comprendere alcuni aspetti del middleware per il messaging
  - introdurre JMS
- Argomenti
  - messaging
  - JMS

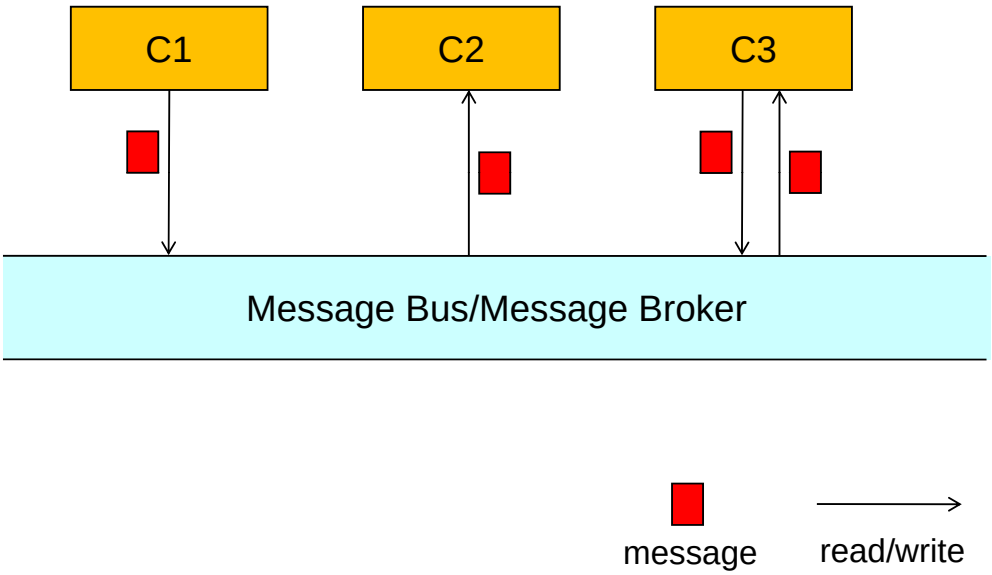


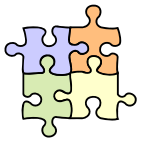
* Messaging

- Il **messaging** è una tipologia di comunicazione event-based
  - consente a *componenti produttori* di inviare *messaggi* a *componenti consumatori*
    - in modo indiretto – tramite un *bus per messaggi*
    - in modo *asincrono*
  - la comunicazione viene avviata dal componente (*produttore*) che produce il messaggio
  - un messaggio non è inviato direttamente al componente (*consumatore*) che consumerà il messaggio
    - piuttosto, un produttore invia i suoi messaggi ad un *bus per messaggi* – che definisce delle *destinazioni* intermedie
  - in modo asincrono, un componente (*consumatore*) legge messaggi da queste destinazioni intermedie
  - il messaging è supportato da middleware opportuno



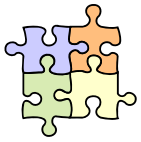
Messaging



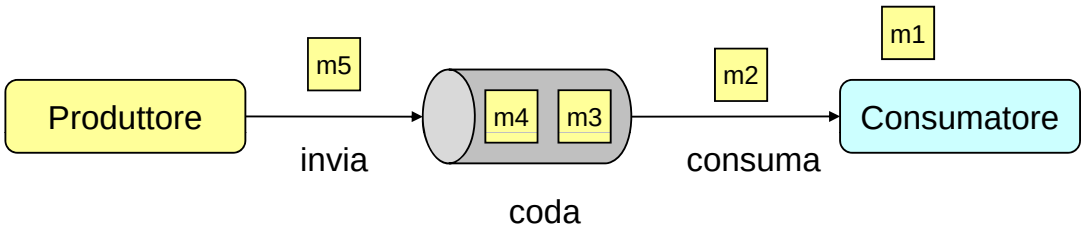


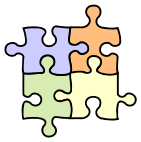
Destinazioni intermedie

- Gli strumenti di messaging forniscono due tipi principali di destinazioni intermedie
  - *coda*
    - ciascun messaggio è consumato da uno ed un solo consumatore
    - è un canale di comunicazione uno-a-uno
  - *topic* (argomento)
    - un messaggio può essere consumato da più consumatori, registrati presso il canale
    - è un canale publisher-suscriber

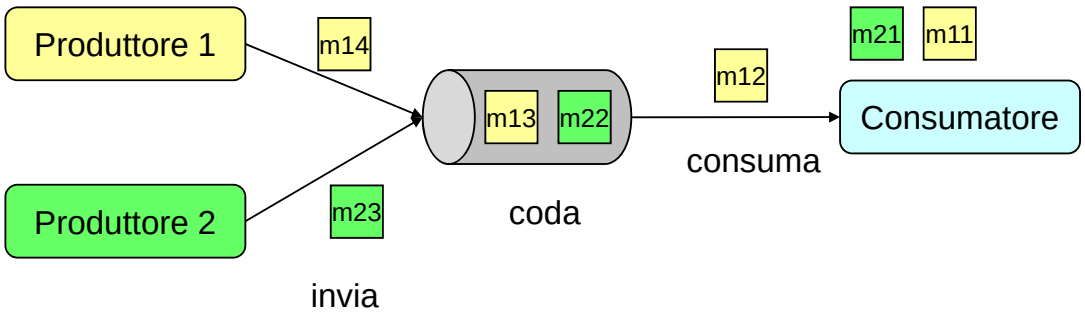


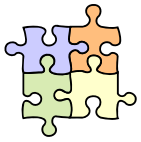
Messaging – code



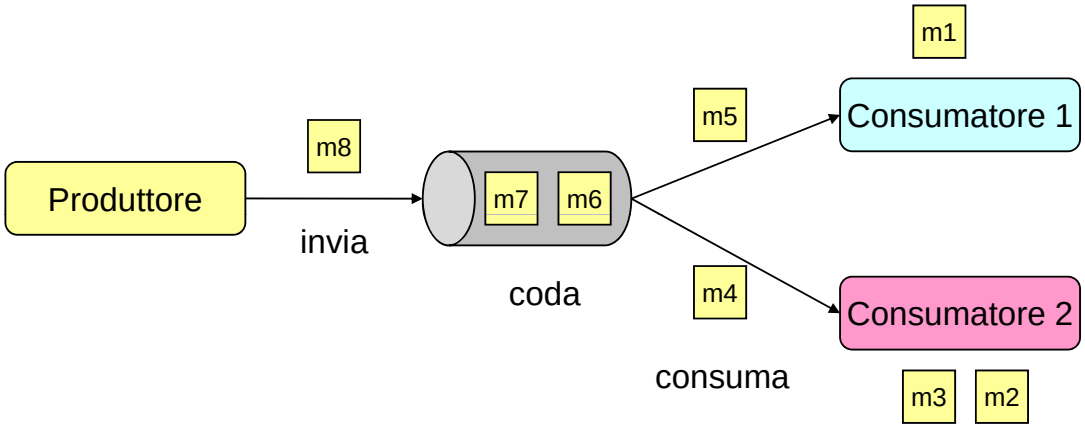


Code: più produttori

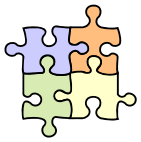




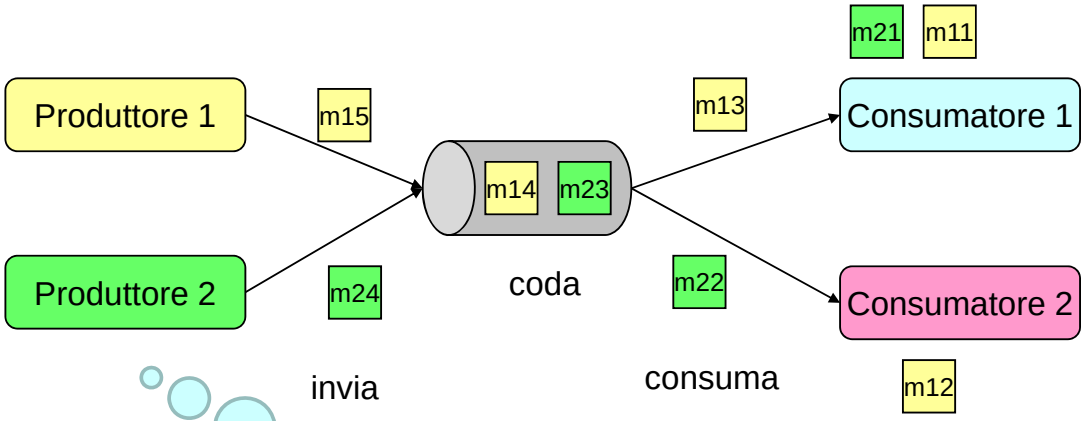
Code: più consumatori



in questo caso, ciascun messaggio viene consumato da un solo consumatore

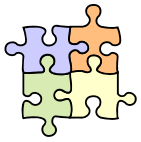


Code: più produttori/più consumatori

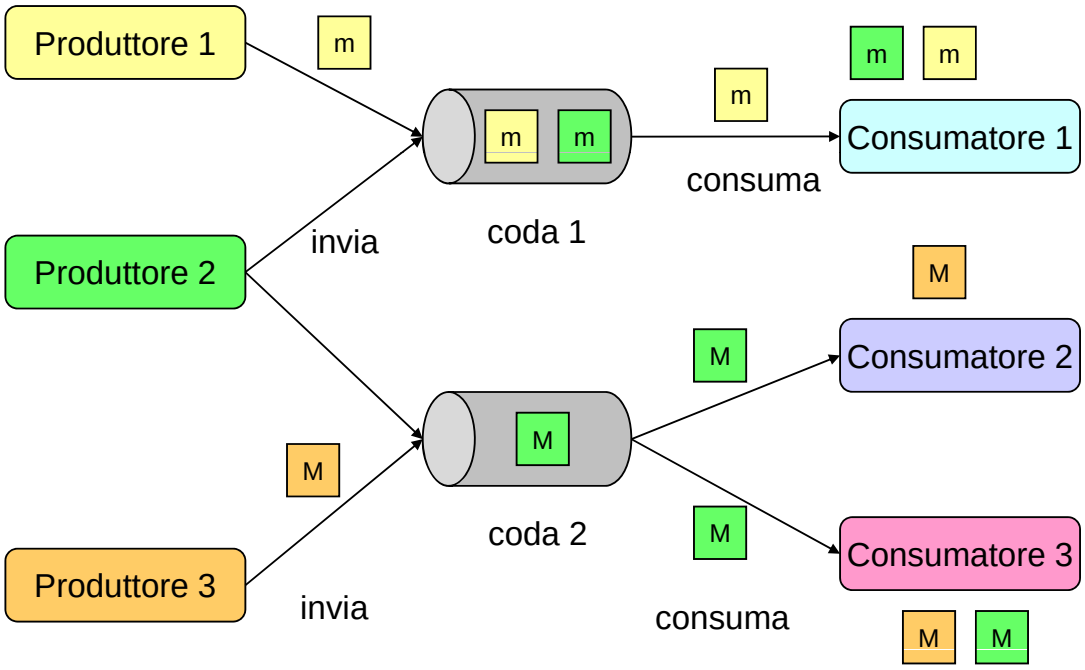


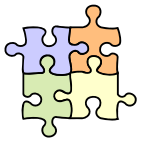
anche in questo caso, ciascun messaggio viene consumato da un solo consumatore

messaggi di uno stesso produttore possono essere consumati da consumatori diversi

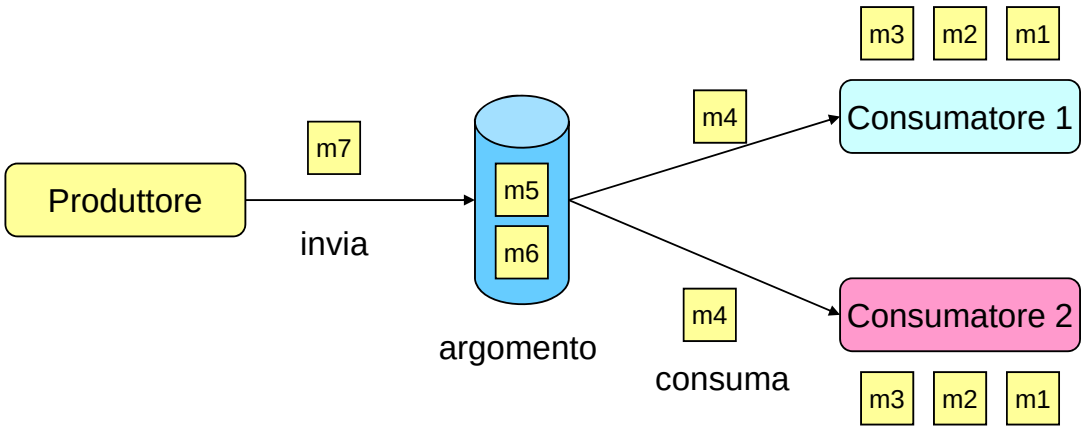


Code: più destinazioni

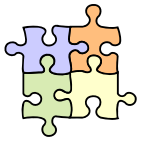




Messaging: argomenti

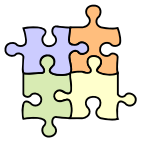


in questo caso, i messaggi possono essere consumati da più consumatori



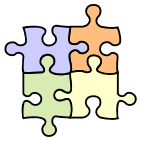
Messaging

- Un sistema di messaging offre una facilitazione *peer-to-peer*
  - un componente può inviare messaggi e ricevere messaggi da ogni altro componente
    - un componente che invia un messaggio ad una destinazione è un *produttore* di messaggi
    - un *consumatore* di messaggi è un componente che accede ad un messaggio da una destinazione
  - per comunicare, un produttore ed un consumatore devono essere d'accordo su quale destinazione usare, e sul formato dei messaggi scambiati
    - il produttore non ha bisogno di conoscere nient'altro del consumatore – e viceversa
    - in particolare, il produttore non deve conoscere l'identità del consumatore – e viceversa



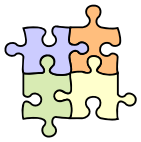
Messaging e comunicazione asincrona

- Un sistema di messaging offre una facilitazione *asincrona*
  - per scambiarsi messaggi, un produttore ed un consumatore non devono essere attivi/disponibili contemporaneamente
    - grazie all'indirezione asincrona realizzata dalla destinazione
    - il produttore può inviare messaggi anche se il consumatore non è attivo
    - il consumatore può ricevere messaggi anche se il produttore non è più attivo
  - questo è diverso da tecnologie sincrone come RMI, in cui un oggetto client e un oggetto servente, per comunicare, devono essere contemporaneamente attivi



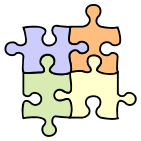
Messaging e posta elettronica

- Anche se sussistono delle analogie tra messaging e posta elettronica, il messaging è diverso dal servizio di posta elettronica (email)
  - la posta elettronica è un meccanismo di comunicazione tra persone, o tra un'applicazione e delle persone
  - il messaging è un meccanismo di comunicazione tra applicazioni e/o componenti software



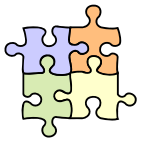
Messaging e accoppiamento

- Complessivamente, il messaging abilita una comunicazione distribuita basata su un *accoppiamento debole* tra produttori e consumatori
  - che devono conoscere destinazione e formato dei messaggi scambiati
  - ma non devono conoscersi ulteriormente
    - in particolare, non è necessaria nessuna conoscenza reciproca relativamente all'interfaccia procedurale dei componenti che devono comunicare
  - non devono nemmeno essere attivi contemporaneamente



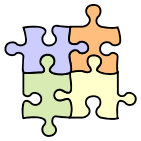
Messaging server-centrico e distribuito

- Alcuni sistemi di messaging sono server-centrici
  - nel senso che le destinazioni sono gestite da un application server centralizzato
  - per scrivere/leggere sulle/dalle destinazioni, l'application server deve essere attivo
- Altri sistemi di messaging sono invece distribuiti
  - sull'host di ciascun componente girano degli opportuni demoni, e le destinazioni sono gestite in modo distribuito



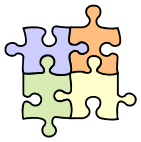
Messaging ed affidabilità

- Un sistema di messaging può offrire un servizio di consegna di messaggi a diversi livelli di *affidabilità*
  - è possibile garantire che ciascun messaggio venga consegnato una ed una sola volta
    - eventualmente anche con un supporto transazionale
  - è anche possibile selezionare una consegna non affidabile di messaggi
    - in cui i messaggi possono perdersi oppure scadere oppure essere ricevuti più volte
  - il livello di affidabilità può essere configurato in modo dipendente anche dal contesto applicativo



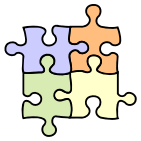
Quando usare un sistema di messaging?

- Quando preferire un sistema di messaging ad un meccanismo di RPC/RMI per far comunicare dei componenti?
  - se l'applicazione deve poter essere eseguita anche se i componenti non sono tutti attivi contemporaneamente
  - se i componenti possono essere progettati in modo tale da inviare informazioni ad altri componenti – continuando a lavorare anche senza ricevere una risposta immediata
  - se i componenti devono/possono essere progettati in modo tale poter ignorare le interfacce degli altri componenti, stabilendo un protocollo di comunicazione basato solo sul formato dei messaggi scambiati
    - se necessario, questo consente la sostituibilità dei componenti produttori e consumatori



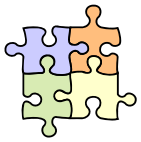
* JMS

- *Java Message Service* (*JMS*)
  - è un'API di Java (di Java EE) che consente alle applicazioni di creare, inviare, ricevere e leggere messaggi
    - definisce un insieme di interfacce – con una relativa semantica – che consente alle applicazioni Java di comunicare mediante un servizio di messaging
    - orientata alla semplicità (minimizzazione dei concetti da apprendere) ed alla portabilità (tra piattaforme Java EE)
    - soluzione server-centrica
- Il servizio di messaging è offerto da un application server Java EE – che non fa parte delle API
  - ad es., Sun AS oppure IBM WebSphere AS



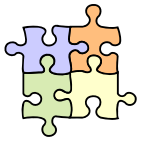
JMS – concetti di base (1)

- Un'applicazione JMS è composta dalle seguenti parti
  - un *provider JMS* – un sistema di messaging che implementa le interfacce JMS e fornisce funzionalità di amministrazione e controllo
    - ad es., gli AS che implementano la piattaforma Java EE comprendono un provider JMS
    - nel mondo .NET, il sistema operativo Windows è un provider di servizi di messaging



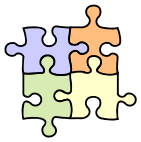
JMS – concetti di base (2)

- *oggetti amministrati* – sono oggetti JMS pre-configurati creati da un amministratore per essere usati dai client
  - in particolare, *destinazioni* (code oppure argomenti) e *factory per le connessioni*
  - l'insieme degli oggetti amministrati può essere considerato l'analogo dello schema di una base di dati relazionale
  - attenzione – le caratteristiche offerte e la relativa semantica possono essere diverse da provider a provider, anche in modo significativo
- *strumenti di amministrazione*
  - ad es., per creare/configurare le destinazioni
- *un servizio di directory JNDI*
  - per la registrazione e l'accesso a risorse mediante nomi simbolici

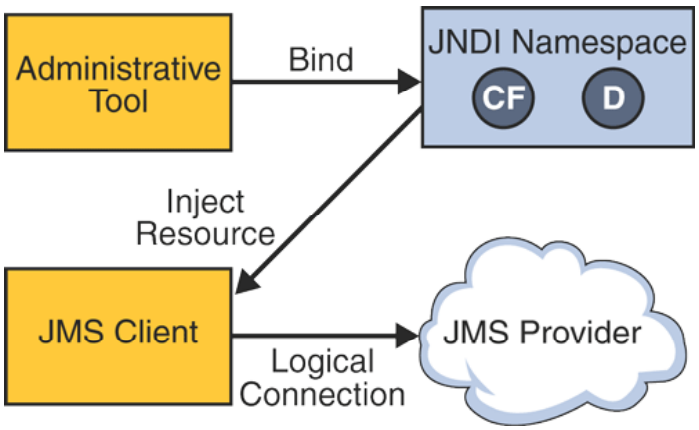


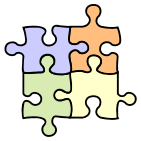
JMS – concetti di base (3)

- uno o più *client JMS* – i programmi o componenti, scritti in Java, che producono e consumano messaggi
  - ad es., un qualunque componente applicativo Java EE, oppure un application client Java EE
- i *messaggi* – sono oggetti che rappresentano informazioni scambiate dai client JMS



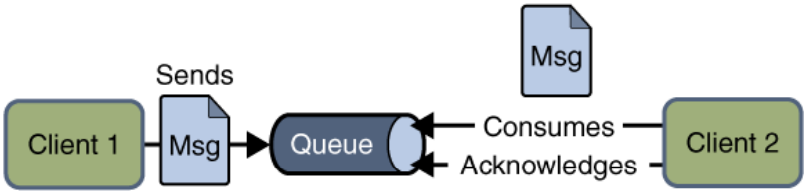
Architettura per JMS

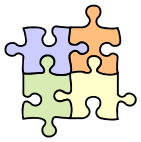




Destinazioni: code

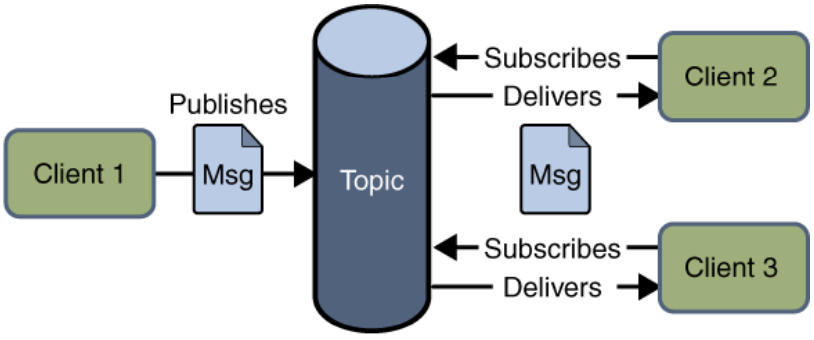
- *Code* – *point-to-point*
  - un client produttore indirizza un messaggio ad una coda specifica
  - un client consumatore di messaggi estrae i messaggi dalla coda stabilita per lo scambio di messaggi col consumatore
  - un messaggio viene consumato da uno ed un solo consumatore

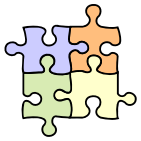




Destinazioni: argomenti

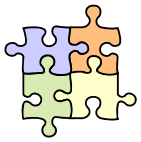
- *Argomenti* (*topic*) – *publisher-subscriber*
  - un produttore indirizza un messaggio ad un topic specifico
  - un client consumatore si può registrare dinamicamente a diversi topic – e riceve notifica dei messaggi inviati al topic – limitatamente al periodo di tempo in cui è registrato al topic
  - un messaggio può essere ricevuto da zero a molti consumatori





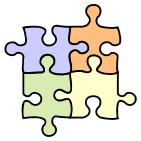
Comunicazione point-to-point

- La comunicazione *point-to-point*
  - basata sui concetti di *coda di messaggi* (*queue*), *sender* e *receiver*
  - ciascun messaggio è indirizzato da un sender ad una coda specifica
  - i clienti receiver estraggono i messaggi dalla coda stabilita
  - la coda mantiene i messaggi fino a quando non sono stati tutti consumati oppure il messaggio “scade”
  - ogni messaggio viene consumato da un solo receiver
    - questo è vero anche se una coda ha più receiver
  - non ci sono dipendenze temporali tra sender e receiver



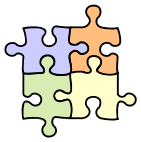
Comunicazione publisher/subscriber

- Nella comunicazione *publisher/subscriber*
  - i client publisher indirizzano i loro messaggi ad un *topic* (*argomento*)
  - i client subscriber di messaggi si registrano dinamicamente ai topic di interesse
  - quando un publisher invia un messaggio ad un topic, il sistema di messaging si occupa della distribuzione dei messaggi a tutti i subscriber registrati
  - il topic mantiene i messaggi solo per il tempo necessario a trasmetterli ai subscriber attualmente registrati
  - un messaggio più avere più consumatori
  - c’è una dipendenza temporale tra publisher e subscriber
    - un subscriber riceve messaggi per un topic solo per il tempo in cui vi è registrato



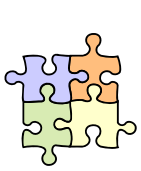
Discussione

- Il messaging point-to-point va usato quando
  - ciascun messaggio deve essere elaborato (con successo) da un solo consumatore
- Il messaging publisher/subscriber va usato quando
  - ciascun messaggio può essere elaborato da zero, uno o più consumatori

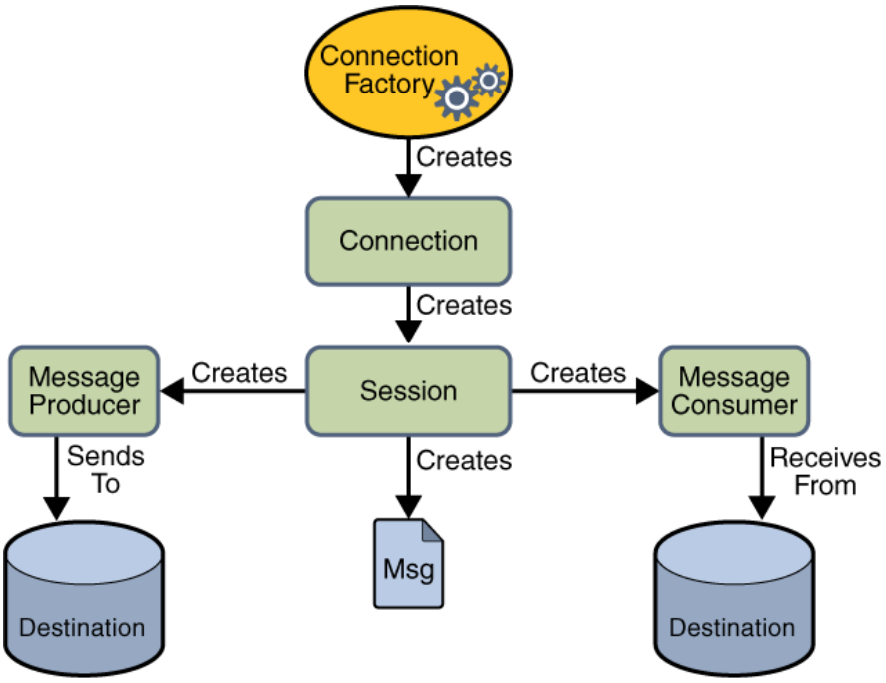


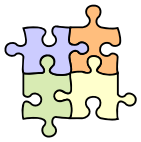
API di JMS (1)

- JMS fornisce
  - interfacce e implementazioni per la gestione di code
    - ad es., ***Queue***, ***QueueConnection***, ***QueueSession***, ***QueueSender*** e ***QueueReceiver***
  - interfacce e implementazioni per la gestione di topic
    - ad es., ***Topic***, ***TopicConnection***, ***TopicSession***, ***TopicPublisher*** e ***TopicSubscriber***
  - ma anche interfacce e implementazioni che generalizzazione questi concetti
    - ad es., ***Destination***, ***Connection***, ***Session***, ***MessageProducer*** e ***MessageConsumer***



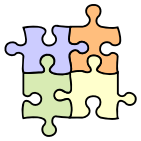
API di JMS (2)





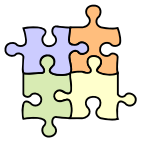
API di JMS (3)

- ***Destination***
  - incapsula una destinazione su un provider
- ***Connection***
  - incapsula una connessione tra un client ed un provider JMS – *in qualche modo analoga ad una connessione JDBC*
- ***Session***
  - un contesto single-threaded per la produzione ed il consumo di messaggi
    - ad es., è una factory di messaggi e destinazioni temporanee
    - single-threaded – come conseguenza, nell'ambito di una sessione, i messaggi sono serializzati, ovvero inviati e/o consumati uno alla volta
  - fornisce un contesto all'elaborazione transazionale dei messaggi – *in qualche modo analoga ad una transazione JDBC*



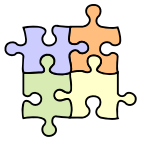
API di JMS (4)

- ***MessageProducer***
  - un oggetto intermediario, di supporto, per inviare messaggi ad una destinazione
  - non è il produttore "logico" del messaggio – ma è un oggetto di supporto necessario
- ***MessageConsumer***
  - un oggetto intermediario, di supporto, per ricevere messaggi da una destinazione
  - non è il produttore "logico" del messaggio – ma è un oggetto di supporto necessario
- ***MessageListener***
  - interfaccia per la ricezione asincrona di messaggi



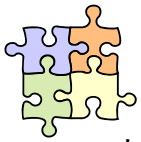
Consumo di messaggi

- Il consumo di messaggi può avvenire secondo due modalità – attenzione, la terminologia è fuorviante!
  - *consumo sincrono* – un consumatore accede ai messaggi da una destinazione mediante un'operazione bloccante di ricezione di un messaggio – "consumo bloccante"
  - *consumo asincrono* – "consumo non bloccante" – un client può registrare un consumatore che implementa un'interfaccia *message listener*
    - deve implementare un metodo ***onMessage*** – che viene eseguito ogni volta che viene ricevuto un messaggio
      - il componente consumatore è un componente applicativo Java EE, e vive in un contenitore Java EE
      - il contenitore Java EE seleziona e notifica ad un componente il fatto che deve consumare un messaggio invocando il metodo ***onMessage***



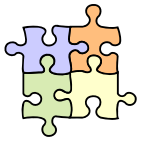
Esempi e configurazione

- Negli esempi che seguono si assume che
  - è installato e configurato un provider JMS
    - ad esempio, Sun Application Server Platform
  - sul provider JMS sono stati definiti e configurati degli opportuni oggetti amministrati
    - una coda fisica jms_MyQueue ed una coda logica jms/MyQueue
    - un argomento logico (jms/MyTopic) ed il corrispondente argomento fisico
    - una factory di connessione, jms/MyConnectionFactory – specifica caratteristiche della modalità d'accesso alla destinazione, ad es., l'affidabilità
  - le applicazioni mostrate sono compilate ed eseguite come **application client** Java EE mediante un IDE opportuno
    - in particolare, Eclipse o NetBeans



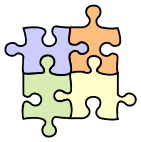
Esempio – produttore di messaggi (1)

```
package simpleproducer;

import javax.jms.*;
import javax.annotation.Resource;

public class Main {

    @Resource(mappedName = "jms/MyQueue")
    private static Queue queue;
    @Resource(mappedName = "jms/MyTopic")
    private static Topic topic;
    @Resource(mappedName = "jms/MyConnectionFactory")
    private static ConnectionFactory connectionFactory;

    public static void main(String[] args) {
        ...
    }
}
```



Annotazioni e iniezione di risorse

- Nell'esempio, l'annotazione @Resource consente di "iniettare" il riferimento ad una risorsa in una variabile
  - quando il compilatore incontra un'annotazione, la riporta, come metadato, nel bytecode – il compilatore non interpreta annotazioni
  - le annotazioni possono essere prese in considerazione dagli strumenti di sviluppo (ad es., JUnit) e dall'ambiente di esecuzione (l'application server in cui viene eseguito un application client)
- Nell'esempio,
  - il valore di queue viene assegnato sulla base di una ricerca JNDI – prima che inizi l'esecuzione delle istruzioni dell'applicazione
  - attenzione, l'iniezione delle risorse avviene solo nella "main class" di un "application client" per Java EE



Annotazioni – in Java

- Le annotazioni non vengono tradotte in istruzioni – ma in "annotazioni" del bytecode, poi interpretate dall'ambiente di sviluppo o di esecuzione
  - spesso semplificano il codice – in particolare, in questo caso mascherano l'accesso al server JNDI per la ricerca delle varie risorse

```
import javax.annotation.Resource;
```

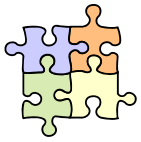
Java EE 5

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

```
import javax.naming.*;    // per jndi
```

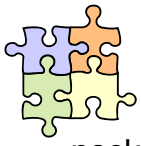
Java EE 1.4

```
String destinationName = "jms/MyQueue";
Context jndiContext = new InitialContext();
Queue queue = (javax.jms.Queue) jndiContext.lookup("jms/MyQueue");
```



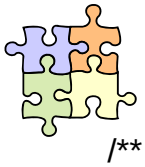
Esempio – produttore di messaggi (2)

```
public static void main(String[] args) {

    /* crea il producer (per la coda) */
    SimpleProducer simpleProducer =
        new SimpleProducer(“Produttore”, queue, connectionFactory);

    /* si connette alla destinazione jms */
    simpleProducer.connect();

    /* invia alcuni messaggi */
    for (int i = 0; i < 10; i++) {
        simpleProducer.sendMessage(“Messaggio #” + i + “ da Produttore”);
    }

    /* chiude la connessione */
    simpleProducer.disconnect();
}
```

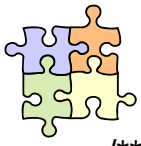
Esempio – produttore di messaggi (3)

```
package simpleproducer;
import javax.jms.*;

public class SimpleProducer {

    /* nome di questo producer */
    private String name;
    /* destinazione di questo producer */
    private Destination destination;
    /* connection factory di questo producer */
    private ConnectionFactory connectionFactory;

    /* connessione jms */
    private Connection connection = null;
    /* sessione jms */
    private Session session = null;

    ...
}
```



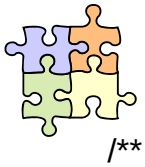
Esempio – produttore di messaggi (4)

```
/** Crea un nuovo SimpleProducer, di nome n, per una Destination d. */
public SimpleProducer(String n, Destination d, ConnectionFactory cf) {
    this.name = n;
    this.destination = d;
    this.connectionFactory = cf;
}
```

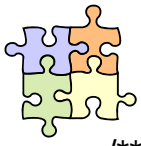
Esempio – produttore di messaggi (5)

```
/** Si connette alla destinazione JMS. */
public void connect() {
    try {
        connection = connectionFactory.createConnection();
        session = connection.createSession(  false, // non transazionale
                                             Session.AUTO_ACKNOWLEDGE);
    } catch (Exception e) {
        System.out.println("Connection problem: " + e.toString());
        disconnect();
    }
}
```



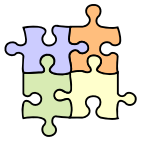
Esempio – produttore di messaggi (6)

```
/** Si disconnette dalla destinazione JMS. */
public void disconnect() {
    if (connection != null) {
        try {
            connection.close();
            connection = null;
        } catch (JMSException e) {
            System.out.println("Disconnection problem: " + e.toString());
        }
    }
}
```

Esempio – produttore di messaggi (7)

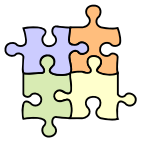
```
/** Invia un messaggio text alla destinazione. */
public void sendMessage(String text) {
    try {
        MessageProducer messageProducer =
                                    session.createProducer(destination);
        TextMessage message = session.createTextMessage();
        message.setText(text);
        System.out.println("Sending message: " + message.getText());
        messageProducer.send(message);
     } catch (JMSException e) {
        System.out.println("MyMessageProducer: " + e.toString());
    }
}
```



Esempio – consumatore sincrono (1)

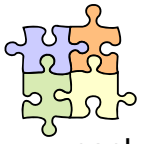
```
package simplesynchconsumer;

import javax.jms.*;
import javax.annotation.Resource;

public class Main {

    @Resource(mappedName = "jms/MyQueue")
    private static Queue queue;
    @Resource(mappedName = "jms/MyTopic")
    private static Topic topic;
    @Resource(mappedName = "jms/MyConnectionFactory")
    private static ConnectionFactory connectionFactory;

    public static void main(String[] args) {
        ...
    }
}
```

come prima



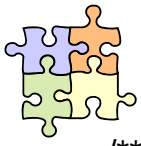
Esempio – consumatore sincrono (2)

```
public static void main(String[] args) {

    /* crea il consumer (per la coda) */
    SimpleSynchConsumer simpleConsumer =
        new SimpleSynchConsumer("Consumatore", queue, connectionFactory);

    /* si connette alla destinazione jms e avvia la ricezione dei messaggi */
    simpleConsumer.connect();
    simpleConsumer.start();

    /* riceve 10 messaggi */
    for (int i=0; i<10; i++) {
        String message = simpleConsumer.receiveMessage();
        System.out.println("Consumatore ha ricevuto: " + message);
    }

    /* termina la ricezione dei messaggi e chiude la connessione */
    simpleConsumer.stop();
    simpleConsumer.disconnect();
}
```



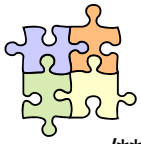
Esempio – consumatore sincrono (3)

```
package simplesynchconsumer;
import javax.jms.*;

public class SimpleSynchConsumer {

    /* nome di questo consumer */
    private String name;
    /* destinazione di questo consumer */
    private Destination destination;
    /* connection factory di questo consumer */
    private ConnectionFactory connectionFactory;

    /* connessione jms */
    private Connection connection = null;
    /* sessione jms */
    private Session session = null;
    /* per la ricezione dei messaggi */
    private MessageConsumer messageConsumer = null;

    ...
}
```

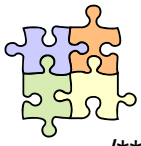
Esempio – consumatore sincrono (4)

```
/** Crea un nuovo SimpleSynchConsumer, di nome n, per una Destination d. */
public SimpleSynchConsumer(String n, Destination d, ConnectionFactory cf) {
    this.name = n;
    this.destination = d;
    this.connectionFactory = cf;
}

... connect e disconnect, come prima ...

...
```



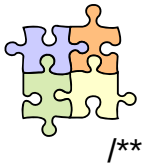
Esempio – consumatore sincrono (5)

```
/** Avvia la ricezione dei messaggi */
public void start() {
    try {
        messageConsumer = session.createConsumer(destination);
        /* avvia la consegna di messaggi per la connessione */
        connection.start();
    } catch (JMSException e) {
        System.out.println("SimpleSynchConsumer: " + e.toString());
        System.exit(1);
    }
}
```

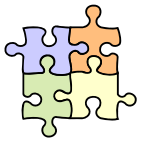
Esempio – consumatore sincrono (6)

```
/** Arresta la ricezione dei messaggi */
public void stop() {
    try {
        /* arresta la consegna di messaggi per la connessione */
        connection.stop();
    } catch (JMSException e) {
        System.out.println("SimpleSynchConsumer: " + e.toString());
        System.exit(1);
    }
}
```



Esempio – consumatore sincrono (7)

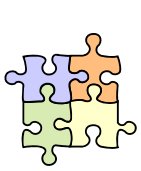
```
/** Riceve un messaggio dalla destinazione */
public String receiveMessage() {

   try {
      Message m = messageConsumer.receive();  // bloccante
      if (m instanceof TextMessage) {
         TextMessage message = (TextMessage) m;
         return message.getText();
      }
   } catch (JMSException e) {
      System.out.println("MySynchMessageConsumer: " + e.toString());
      System.exit(1);
   }
   return null;
}
```

Esempio – consumatore asincrono (1)

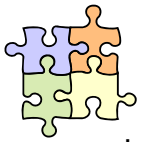
```
package simpleasynchconsumer;

import javax.jms.*;
import javax.annotation.Resource;

public class Main {

   @Resource(mappedName = "jms/MyQueue")
   private static Queue queue;
   @Resource(mappedName = "jms/MyTopic")
   private static Topic topic;
   @Resource(mappedName = "jms/MyConnectionFactory")
   private static ConnectionFactory connectionFactory;

   public static void main(String[] args) {
      ...
   }
}
```

come prima



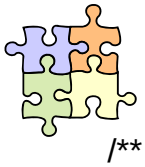
Esempio – consumatore asincrono (2)

```
public static void main(String[] args) {

    /* crea il consumer (per la coda) */
    SimpleAsynchConsumer simpleConsumer =
        new SimpleAsynchConsumer("Consumatore", queue, connectionFactory);

    /* si connette alla destinazione jms */
    simpleConsumer.connect();

    /* riceve messaggi */
    simpleConsumer.receiveMessages();

    /* chiude la connessione */
    simpleConsumer.disconnect();
}
```

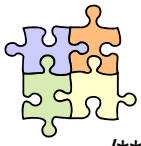
Esempio – consumatore asincrono (3)

```
package simpleasynchconsumer;

import javax.jms.*;

public class MyAsynchMessageConsumer {
    /* nome di questo consumer */
    private String name;
    /* destinazione di questo consumer */
    private Destination destination;
    /* connection factory di questo consumer */
    private ConnectionFactory connectionFactory;

    /* connessione jms */
    private Connection connection = null;
    /* sessione jms */
    private Session session = null;
    /* per la ricezione dei messaggi */
    private MessageConsumer messageConsumer = null;

    ...
}
```



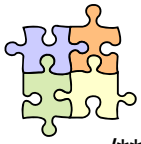
Esempio – consumatore asincrono (4)

```
/** Crea un nuovo SimpleAsynchConsumer, di nome n, per una Destination d. */
public SimpleAsynchConsumer(String n, Destination d, ConnectionFactory cf) {
    this.name = n;
    this.destination = d;
    this.connectionFactory = cf;
}

... connect e disconnect, come prima ...

...
```

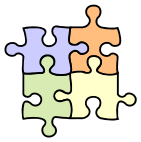
Esempio – consumatore asincrono (5)

```
/** Riceve messaggi dalla destinazione */
public void receiveMessages() {
    try {
        MessageConsumer messageConsumer =
                                        session.createConsumer(destination);
        /* crea l'ascoltatore di messaggi */
        TextListener textListener = new TextListener(this);
        messageConsumer.setMessageListener(textListener);
        /* avvia la consegna di messaggi */
        connection.start();
        while (true) {}    // aspetta messaggi
        /* termina la consegna di messaggi */
        connection.stop();
    } catch (JMSException e) { ...}
}
```



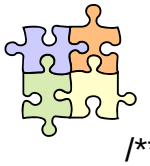
Esempio – consumatore asincrono (6)

```
/** Ricezione di un messaggio. */
public void messageReceived(String msg) {
    System.out.println("Reading message: " + msg);
}
```

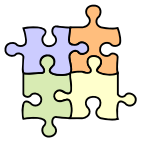
Esempio – consumatore asincrono (7)

```
package simpleasynchconsumer;

import javax.jms.*;

public class TextListener implements MessageListener {

   /* consumer di questo TextListener */
   SimpleAsynchConsumer consumer;

   /** Crea un nuovo TextListener per il consumer c. */
   public TextListener(SimpleAsynchConsumer c) {
       this.consumer = c;
   }

   /** Riceve un messaggio */
   public void onMessage(Message m) {
       ...
   }
}
```



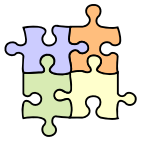
Esempio – consumatore asincrono (8)

```
/** Riceve un messaggio */
public void onMessage(Message m) {
    if (m instanceof TextMessage) {
        TextMessage  message = (TextMessage) m;
        try {
            consumer.messageReceived(message.getText());
        } catch (JMSException e) {
            System.out.println("TextListener.onMessage(): " + e.toString());
        }
    }
}
```

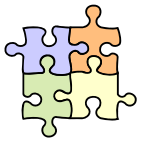
Osservazioni

- Sicuramente possibile fare numerose modifiche e miglioramenti al codice – ad esempio
  - classi di utilità per separare/mettere a fattor comune servizi non legati allo scambio di messaggi
  - comportamento del consumatore basato sul contenuto dei messaggi
    - la ragion d'essere del messaging
  - uso di un messaggio che indica la "fine della sessione"
    - per consentire la terminazione dei consumatori



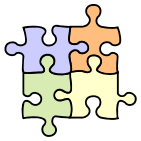
- Alcuni esperimenti con le code

- Esperimento A con le code
  - avvio il consumatore, poi avvio il produttore che invia N messaggi
  - conseguenze
    - il produttore invia N messaggi
    - il consumatore riceve N messaggi



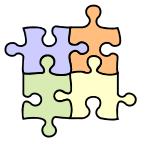
Alcuni esperimenti con le code

- Esperimento B con le code
  - il consumatore non è inizialmente attivo, il produttore invia N messaggi
    - il produttore invia N messaggi, e termina
  - poi viene avviato il consumatore
  - conseguenze
    - il consumatore riceve N messaggi



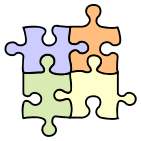
Alcuni esperimenti con le code

- Esperimento C con le code
  - avvio il consumatore, e due produttori sulla stessa coda
  - conseguenze
    - un produttore invia N messaggi
    - un produttore invia M messaggi
    - il consumatore riceve N+M messaggi



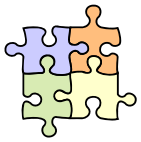
Alcuni esperimenti con le code

- Esperimento D con le code
  - avvio due consumatori sulla stessa coda, poi il produttore invia N messaggi
  - conseguenze
    - il produttore invia N messaggi
    - un consumatore riceve X messaggi – l'altro consumatore riceve gli altri N-X messaggi
- Attenzione
  - questo esperimento non è supportato nello stesso modo da tutti gli application server
    - nessun problema con WebSphere AS
    - da configurare con Sun AS
    - con Sun AS sembra non essere possibile avere più di due consumatori sulla stessa coda



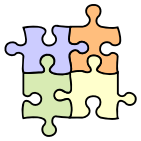
Alcuni esperimenti con gli argomenti

- Esperimento A con gli argomenti
  - avvio un consumatore, poi avvio il produttore che invia N messaggi
  - conseguenze
    - il produttore invia N messaggi
    - il consumatore riceve N messaggi



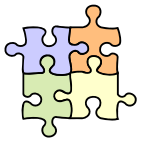
Alcuni esperimenti con gli argomenti

- Esperimento B con gli argomenti
  - il consumatore non è inizialmente attivo, il produttore invia N messaggi
    - il produttore invia N messaggi, e termina
  - poi viene avviato il consumatore
  - conseguenze
    - il consumatore non riceve nessun messaggio



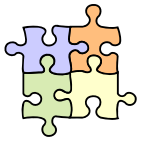
Alcuni esperimenti con gli argomenti

- Esperimento C con gli argomenti
  - avvio un consumatore, e due produttori sullo stesso argomento
  - conseguenze
    - un produttore invia N messaggi
    - un produttore invia M messaggi
    - il consumatore riceve N+M messaggi



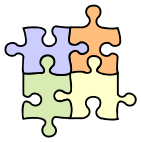
Alcuni esperimenti con gli argomenti

- Esperimento D con gli argomenti
  - avvio due consumatori sullo stesso argomento, poi un produttore invia N messaggi
  - conseguenze
    - il produttore invia N messaggi
    - ciascun consumatore riceve N messaggi



- Ulteriori considerazioni

- Chi invoca onMessage()?
- Per realizzare uno scambio di messaggi richiesta/risposta
  - viene normalmente usata una coda per le richieste ed una coda separata per le risposte
- Come gestire il caso in cui N produttori inviano richieste ad M consumatori – ma poi vogliono delle risposte indirizzate espressamente a loro?
  - è possibile creare delle destinazioni temporanee
  - un messaggio di richiesta può contenere un campo ReplyTo – specifica dove inviare la risposta a quel messaggio



Informazioni associate ai messaggi

- Ad un messaggio possono essere associate diverse informazioni utili
  - un message ID
  - un correlation ID
    - ad es., un messaggio di risposta indica come correlation ID il message ID della richiesta
  - la destinazione a cui è stato inviato
  - la destinazione ReplyTo a cui vanno inviate eventuali risposte
    - potrebbe essere una destinazione temporanea
  - il tempo di Expiration
  - una modalità di consegna – persistente, non persistente