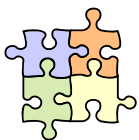


Architetture Software

Messaging (middleware)

Dispensa ASW 840

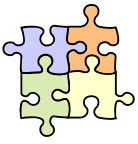
ottobre 2011



- Fonti

- [Hohpe&Woolf 2004] Enterprise Integration Patterns
 - <http://www.enterpriseintegrationpatterns.com/>
 - <http://eaipatterns.com/>

- The Java EE 6 Tutorial
 - <http://download.oracle.com/javasee/6/docs/tutorial/doc/>
 - Java Message Service Concepts
 - Java Message Service Examples



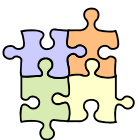
- Obiettivi e argomenti

□ Obiettivi

- comprendere alcuni aspetti del messaging – qui inteso come tecnologia di middleware
- introdurre Java Message Service (JMS)

□ Argomenti

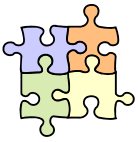
- messaging
- JMS



* Messaging

□ Il **messaging** è una tecnologia di comunicazione per sistemi distribuiti

- i componenti o le applicazioni comunicano, in una relazione tra pari (peer-to-peer), scambiandosi messaggi
 - un *componente* (nel ruolo di *produttore*) può inviare *messaggi* a un altro *componente* (nel ruolo di *consumatore*)
 - questa modalità di comunicazione è supportata da opportuni strumenti di middleware
-
- Il messaging è un paradigma di comunicazione significativamente diverso da quello richiesta/risposta su cui sono basati RPC e RMI
 - con RPC/RMI, la comunicazione è iniziata dal consumatore, è diretta ed è sincrona
 - nel messaging, la comunicazione è iniziata dal produttore, è indiretta ed è asincrona



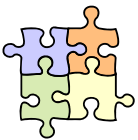
Caratteristiche del messaging

- La comunicazione viene avviata dal componente (*produttore*) che produce il messaggio
 - e non dal componente (*consumatore*) che utilizzerà le informazioni contenute nel messaggio
- La comunicazione è *indiretta*
 - il produttore non invia il messaggio direttamente al consumatore del messaggio
 - piuttosto, il produttore invia i suoi messaggi a un canale di comunicazione intermedio – chiamato *bus per messaggi*
- La comunicazione è *asincrona*
 - il consumatore legge messaggi da destinazioni intermedie
 - questo può avvenire anche in un momento diverso da quando il messaggio è stato inviato – ovvero, non necessariamente nello stesso momento in cui il messaggio viene inviato

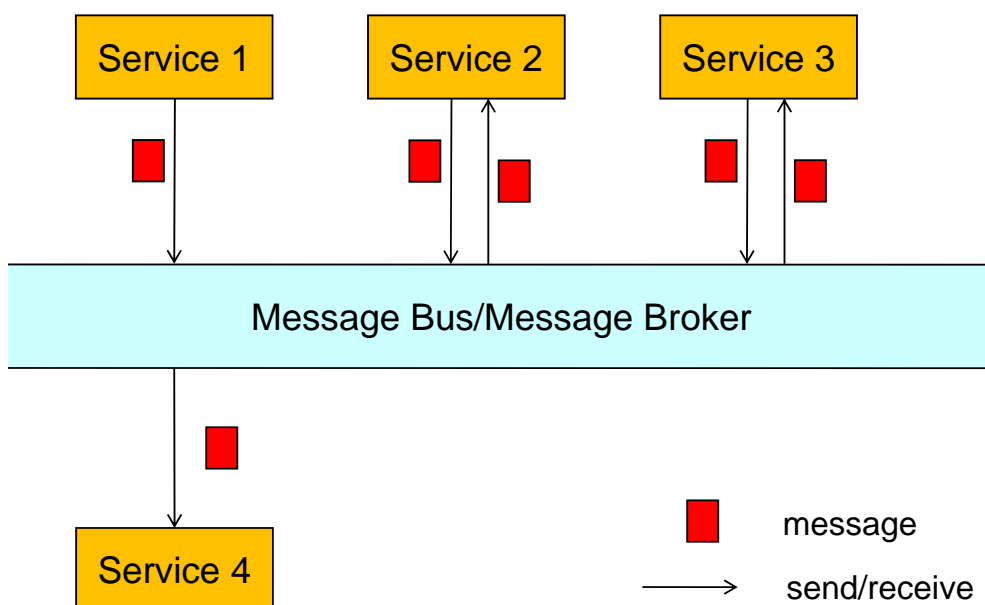
5

Messaging (middleware)

Luca Cabibbo – ASw



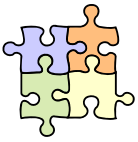
Un sistema di messaging



6

Messaging (middleware)

Luca Cabibbo – ASw



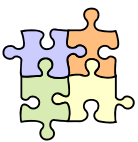
Destinazioni intermedie

- In un sistema di messaging, i messaggi non vengono scambiati genericamente su un “indistinto” message bus
 - piuttosto, i messaggi vengono scambiati mediante *destinazioni intermedie* – o *canali per messaggi*
 - ciascuna destinazione intermedia è identificata univocamente – mediante un nome, appunto, univoco

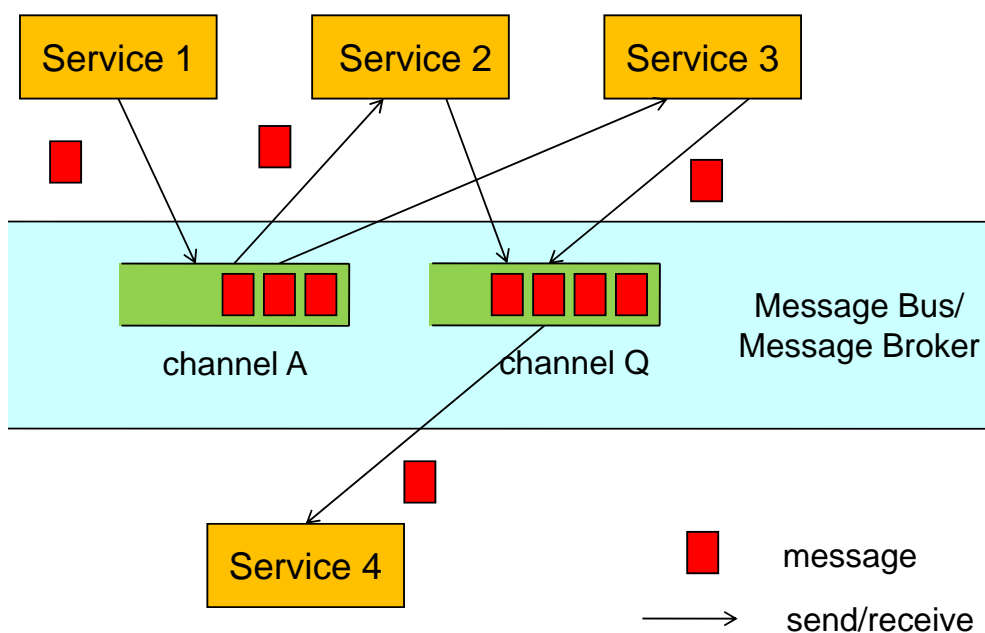
7

Messaging (middleware)

Luca Cabibbo – ASw



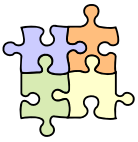
Un sistema di messaging



8

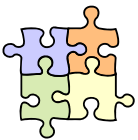
Messaging (middleware)

Luca Cabibbo – ASw

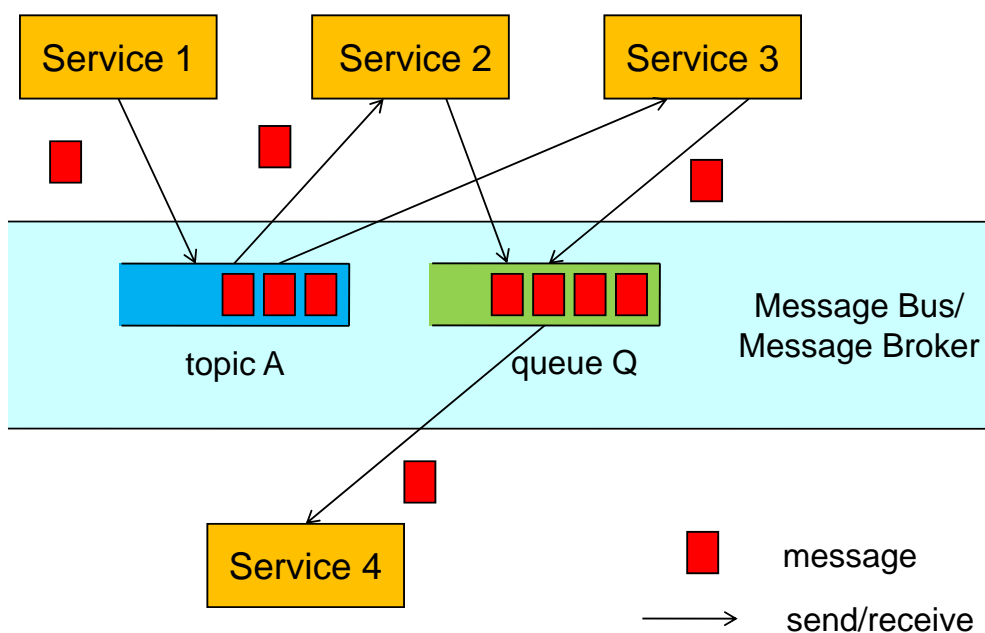


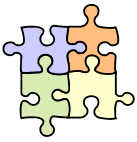
Destinazioni intermedie

- Inoltre, esistono due tipi principali di destinazioni intermedie
 - *queue – coda*
 - un messaggio inviato a una coda sarà consumato da uno ed un solo consumatore
 - è dunque un canale di comunicazione “a-uno”
 - *topic – argomento*
 - un messaggio inviato a un argomento può essere consumato da più consumatori, registrati presso la destinazione
 - è un canale “a-molti”, di tipo publisher-subscriber



Un sistema di messaging





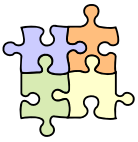
Produttori, consumatori, destinazioni

- In generale, in un'applicazione di messaging ci sono
 - una molteplicità di destinazioni intermedie – sia code che topic – ciascuna destinazione è rivolta a veicolare una certa tipologia di messaggi
 - una molteplicità di componenti – ciascun componente può essere produttore di messaggi, di uno o più tipi – ma anche consumatore di messaggi, di uno o più tipo
 - ciascun componente produttore
 - produce messaggi di una o più tipologie – in corrispondenza, li invia ad una o più destinazioni intermedie
 - ciascun componente consumatore
 - consuma messaggi di una o più tipologie – che riceve da una o più destinazioni intermedie
 - un componente può essere sia produttore che consumatore



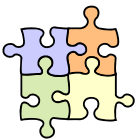
Invocazione implicita

- Il messaging viene anche chiamato *invocazione implicita*
 - quando un consumatore riceve un messaggio, di solito esegue delle azioni – un'“operazione” o “metodo”
 - l'operazione da eseguire viene scelta sulla base del contenuto del messaggio – e non sulla base di una richiesta diretta da parte del produttore del messaggio
 - l'invocazione dell'operazione è, dunque, “implicita”

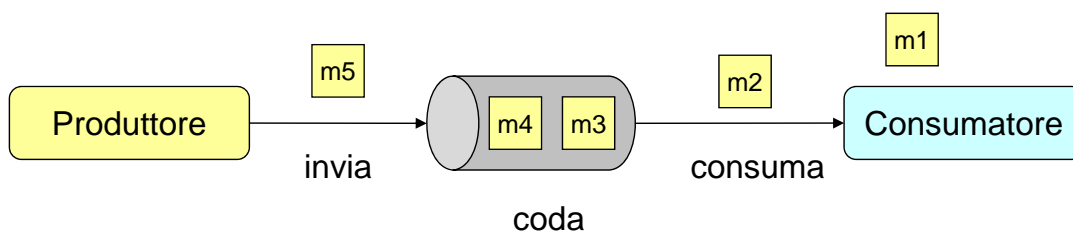


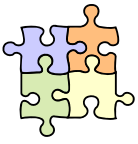
Invocazione implicita

- In un sistema di messaging
 - i produttori vogliono che vengano soddisfatte alcune loro richieste (di un certo tipo)
 - i produttori codificano queste richieste sotto forma di messaggi di un certo tipo, e le inviano a una destinazione opportuna
 - i consumatori sono in grado di gestire richieste (di un certo tipo)
 - i consumatori, quando sono disponibili ad accettare richieste, leggano messaggi da una destinazione opportuna – poi interpretano il messaggio, ed elaborano i dati in esso contenuti eseguendo l'operazione più opportuna
 - un consumatore può avere anche lo scopo di gestire solo parzialmente il messaggio di un produttore – in questi casi, creano un nuovo messaggio e lo inviano ad un'altra destinazione, affinché l'elaborazione prosegua

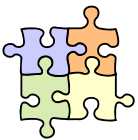
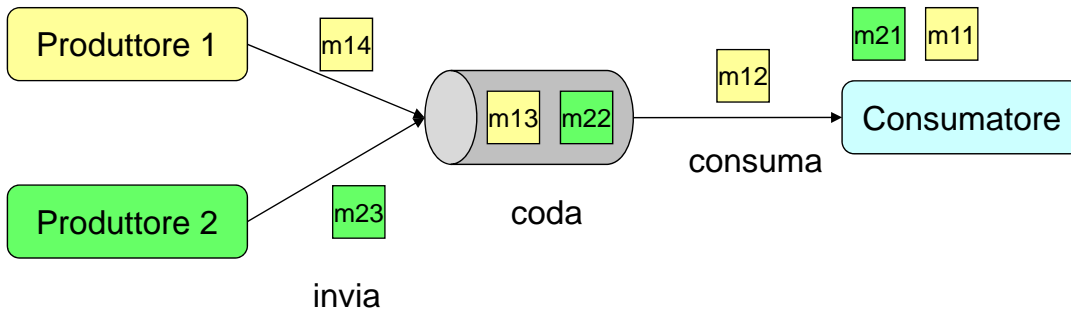


Messaging - code

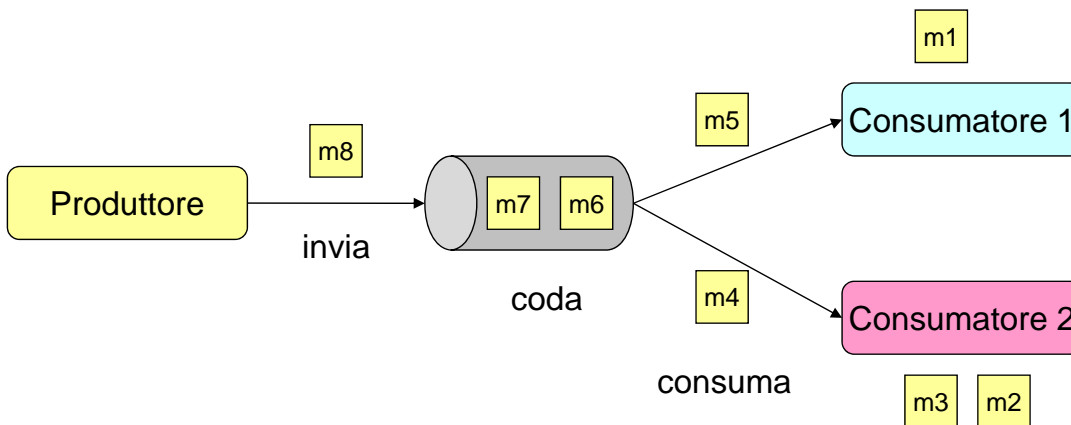




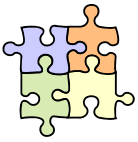
Code: più produttori



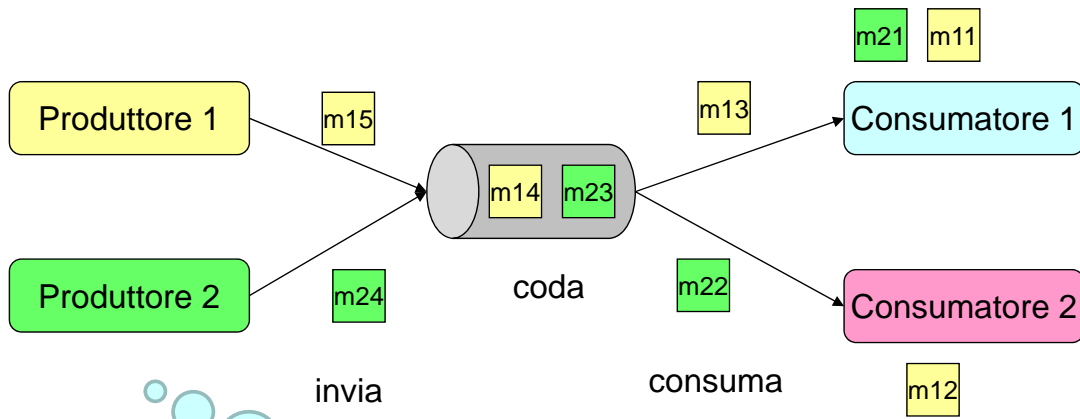
Code: più consumatori



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



Code: più produttori/più consumatori

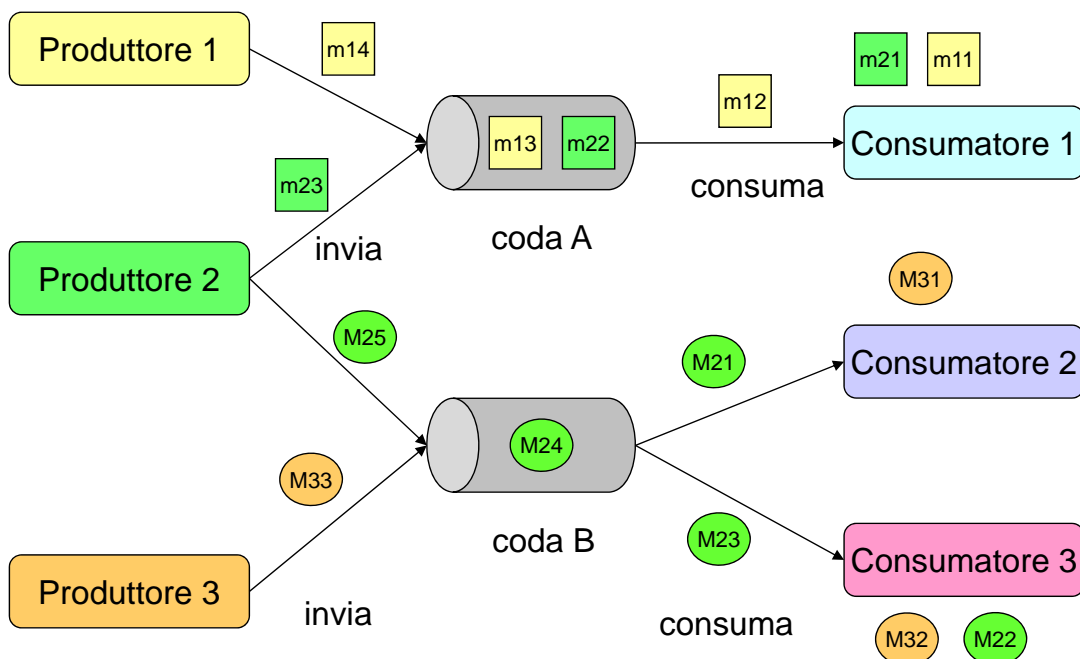


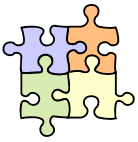
anche in questo caso, ciascun messaggio viene consumato da un solo consumatore

messaggi di uno stesso produttore possono essere consumati da consumatori diversi

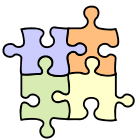
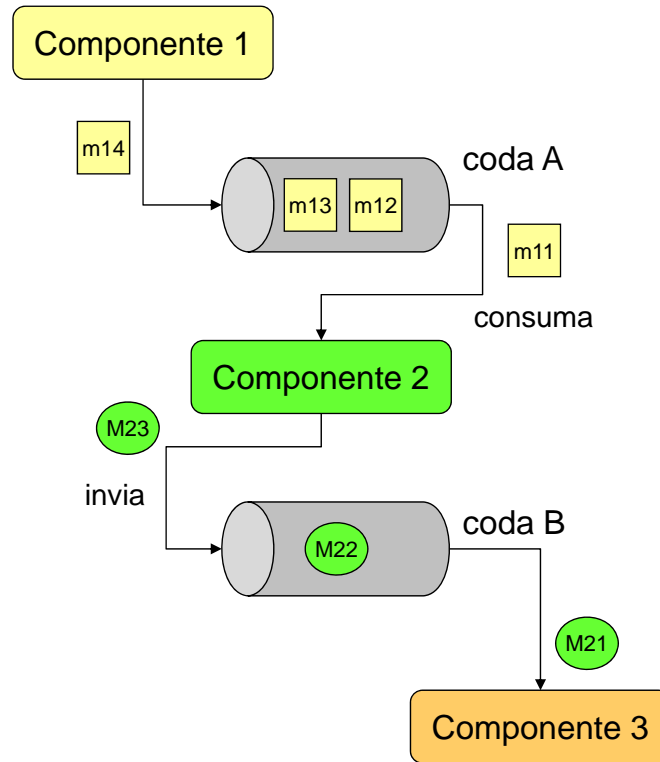


Code: più destinazioni (più tipi di messaggi)

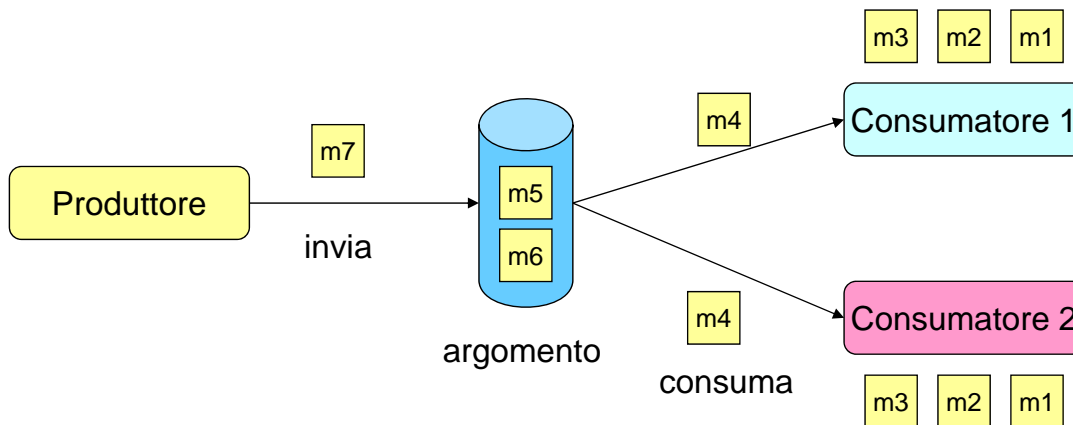




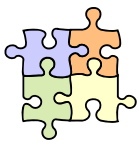
Code: componenti consumatori/produttori



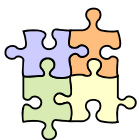
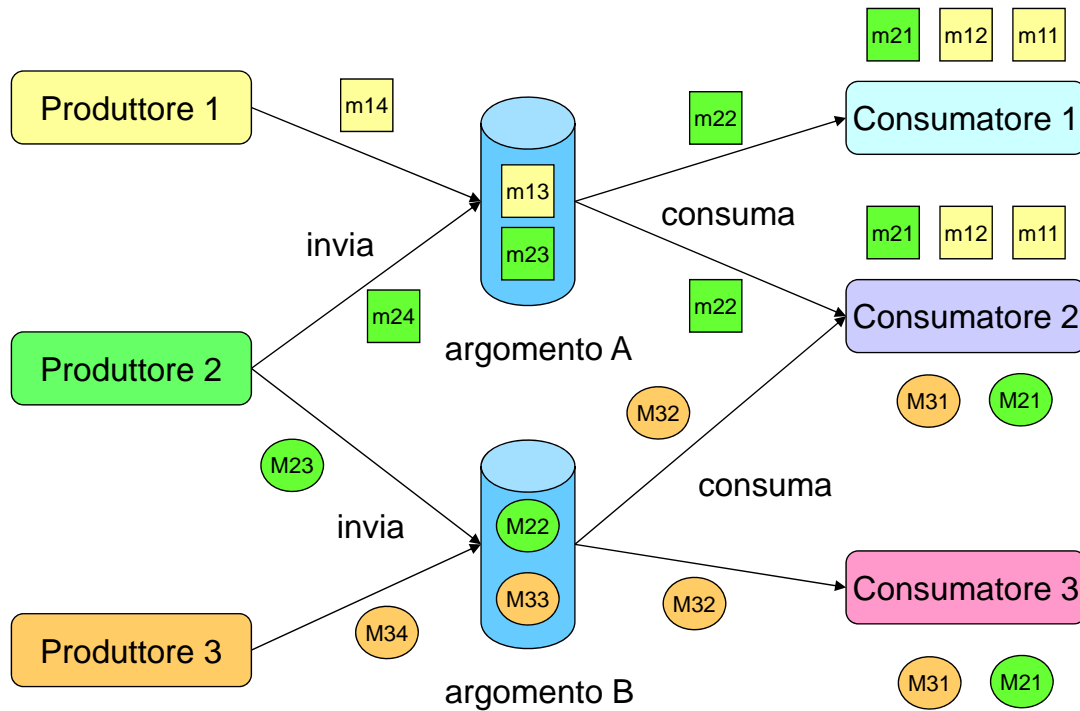
Messaging: argomenti



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

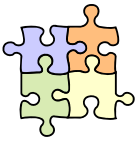


Messaging: argomenti



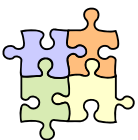
Produttori e consumatori

- Il messaging è una modalità di comunicazione *peer-to-peer*
 - in linea di principio, un componente può inviare messaggi e ricevere messaggi da ogni altro componente
 - attenzione, in realtà i messaggi non sono scambiati direttamente tra componenti – ma vengono scambiati indirettamente mediante l'ausilio delle destinazioni intermedie
 - in un particolare scambio di messaggi
 - il componente che invia un messaggio a una destinazione è il *produttore* del messaggio
 - il componente che accede a un messaggio da una destinazione è un *consumatore* del messaggio



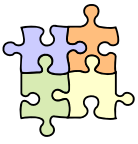
Produttori e consumatori

- Per comunicare, una coppia di componenti (produttore e consumatore) devono essere d'accordo
 - su quale destinazione usare, e
 - sul formato dei messaggi da scambiare
- Il produttore non ha bisogno di conoscere altro del consumatore – e viceversa
 - in particolare, il produttore non deve conoscere l'identità del consumatore – e nemmeno la sua interfaccia “procedurale”, intesa come operazioni che il consumatore sa eseguire



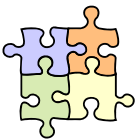
Comunicazione asincrona

- Il messaging è una modalità di comunicazione **asincrona**
 - l'invio dei messaggi avviene con una modalità “send and forget”
 - nello scambio di messaggi, produttore e consumatore non devono essere attivi/disponibili contemporaneamente
 - grazie all'indirizzamento asincrono realizzata dal message bus
 - il produttore può inviare un messaggio anche se il consumatore (inteso come il componente che consumerà quel messaggio) non è (ancora) attivo
 - il consumatore può ricevere un messaggio anche se il produttore (inteso come il componente che ha prodotto il messaggio) non è (più) attivo
 - questo è diverso da tecnologie **sincrone** basate su protocolli richiesta/risposta (ad es., RMI), in cui un oggetto client e un oggetto servente, per comunicare, devono essere attivi contemporaneamente



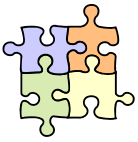
Analogie

- Alcune analogie – riferite alle persone
 - la modalità di comunicazione sincrona è simile alla comunicazione che avviene tra due persone in una telefonata
 - la modalità di comunicazione asincrona (messaging) è simile alla comunicazione che avviene tra due persone nello scambio di messaggi di posta elettronica
 - attenzione, il messaging è diverso dal servizio di posta elettronica
 - 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



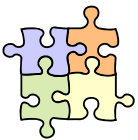
Accoppiamento debole

- Complessivamente, il messaging è una modalità di comunicazione distribuita che sostiene un *accoppiamento debole* tra componenti software
 - nello scambio di messaggi, produttori e consumatori devono conoscere il formato dei messaggi scambiati e la destinazione usata per lo scambio di messaggi
 - ma non devono conoscersi ulteriormente
 - in particolare, non è necessaria nessuna conoscenza reciproca relativamente all'identità, alla locazione, ed all'interfaccia procedurale dei componenti che devono comunicare
 - non devono nemmeno essere attivi contemporaneamente



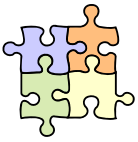
Messaging e affidabilità

- Nei sistemi di messaging, la consegna dei messaggi può essere basata su diversi livelli di **affidabilità** – selezionabili dal programmatore
 - è possibile una consegna non affidabile (di tipo “best effort”) dei messaggi
 - in cui i messaggi possono perdersi oppure scadere oppure essere ricevuti più volte
 - tuttavia, è anche possibile avere una garanzia che ciascun messaggio (inviato a una coda) venga consegnato (o consegnato con successo) una e una sola volta
 - eventualmente anche con un supporto persistente o addirittura transazionale
 - in generale, il livello di affidabilità può essere (e va) configurato in modo dipendente anche dal contesto applicativo



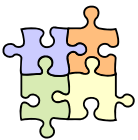
Messaging server-centrico e distribuito

- 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



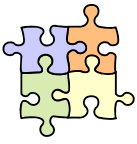
Quando usare un sistema di messaging?

- Quando preferire un sistema di messaging a 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
 - come caso estremo, se i componenti che devono comunicare sono stati sviluppati indipendentemente l'uno dall'altro



* JMS

- **Java Message Service (JMS)**
 - è un'API di Java (più precisamente, di Java EE) che consente alle applicazioni (o componenti) 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) e alla portabilità (tra piattaforme Java EE)
 - soluzione server-centrica
- Il servizio di messaging (vero e proprio) è offerto da un application server Java EE – che non fa parte delle API
 - ad es., Glassfish AS oppure IBM WebSphere AS



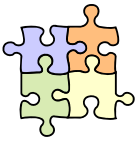
What is messaging?

- ❑ **Messaging** is a method of communication between software components or applications
 - a messaging system is a peer-to-peer facility: a messaging client can send messages to, and receive messages from, any other client
 - each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages
- ❑ Messaging enables distributed communication that is **loosely coupled**
 - a component sends a message to a destination, and the recipient can retrieve the message from the destination
 - however, the sender and the receiver do not have to be available at the same time in order to communicate
 - in fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender
 - the sender and the receiver need to know only which message format and which destination to use
 - in this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods



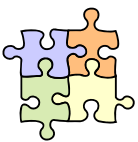
What is the JMS API?

- ❑ The *Java Message Service (JMS)* is a Java API that allows applications to create, send, receive, and read messages
 - the JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations
- ❑ The JMS API minimizes the set of concepts a programmer must learn in order to use messaging products but provides enough features to support sophisticated messaging applications – it also strives to maximize the portability of JMS applications across JMS providers in the same messaging domain
- ❑ The JMS API **enables communication** that is not only **loosely coupled** but also
 - **asynchronous**: a JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them
 - **reliable**: the JMS API can ensure that a message is delivered once and only once – lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages



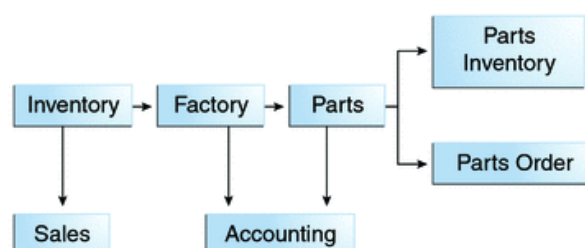
When can you use the JMS API?

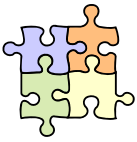
- An enterprise application provider is likely to choose a messaging API over a tightly coupled API, such as remote procedure call (RPC), under the following circumstances
 - the provider wants the components not to depend on information about other components' interfaces, so that components can be easily replaced
 - the provider wants the application to run whether or not all components are up and running simultaneously
 - the application business model allows a component to send information to another and to continue to operate without receiving an immediate response



When can you use the JMS API?

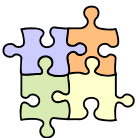
- For example, components of an enterprise application for an automobile manufacturer can use the JMS API in situations like these
 - the inventory component can send a message to the factory component when the inventory level for a product goes below a certain level so that the factory can make more cars
 - the factory component can send a message to the parts components so that the factory can assemble the parts it needs.
 - the parts components in turn can send messages to their own inventory and order components to update their inventories and to order new parts from suppliers
 - both the factory and the parts components can send messages to the accounting component to update their budget numbers
 - the business can publish updated catalog items to its sales force





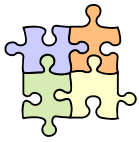
JMS - concetti di base (1)

- Le applicazioni basate su JMS fanno riferimento a un certo numero di concetti ed elementi
 - un *provider JMS* – un sistema di messaging che implementa le interfacce JMS e fornisce funzionalità di amministrazione e controllo
 - in particolare, gli AS che implementano la piattaforma Java EE possono comprendere un provider JMS
 - attenzione, esistono anche implementazioni “parziali” di Java EE – che potrebbero non comprendere un provider JMS
 - la nozione di “provider” di un servizio esiste anche in altri contesti
 - nel mondo .NET, un provider di servizi di messaging può essere il sistema operativo Windows
 - un DBMS può essere considerato un provider di un “servizio di basi di dati”



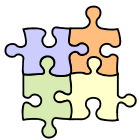
JMS - concetti di base (2)

- Le applicazioni JMS fanno riferimento a un certo numero di concetti ed elementi
 - 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
 - attenzione, il termine “client” può essere fuorviante – in effetti questi componenti agiscono di solito come “peer”
 - i *messaggi* – sono oggetti che rappresentano informazioni scambiate dai client JMS

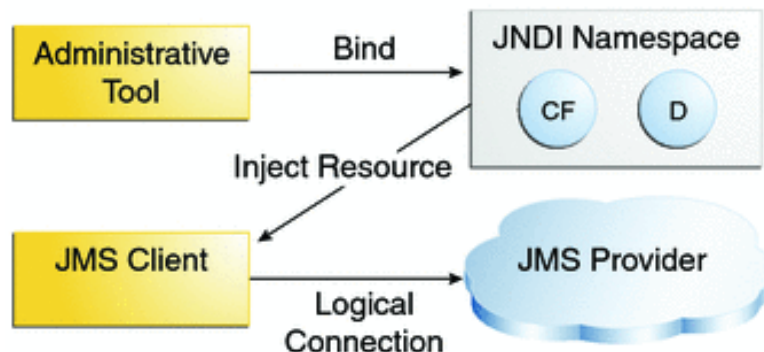


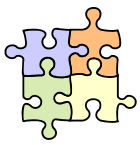
JMS - concetti di base (3)

- Le applicazioni JMS fanno riferimento a un certo numero di concetti ed elementi
 - **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 è l'analogo dello schema di una base di dati relazionale per un DBMS
 - attenzione – le caratteristiche e la semantica di questi oggetti amministrati possono variare da provider a provider
 - **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



Architettura per JMS

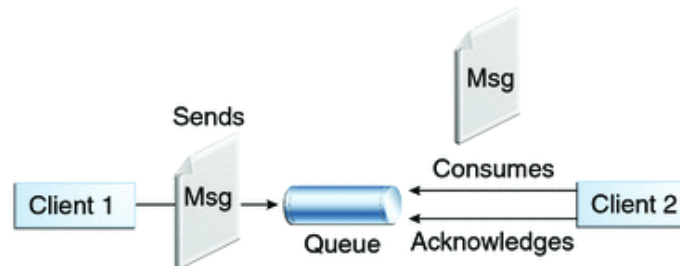




Code

□ *Coda (queue) – point-to-point*

- un client produttore indirizza un messaggio a una coda specifica
- un client consumatore di messaggi estrae i messaggi dalla coda stabilita
- ciascun messaggio viene consumato da uno ed un solo consumatore



- la coda conserva i messaggi che gli sono stati inviati fino a quando questi messaggi non sono stati consumati – oppure non sono “scaduti”

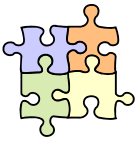


Argomenti

□ *Argomento (topic) – publisher-subscriber*

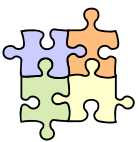
- un produttore indirizza un messaggio ad un topic specifico
- un client consumatore si può registrare dinamicamente a diversi topic
- un client consumatore riceve notifica dei messaggi inviati al topic – ma solo limitatamente al periodo di tempo in cui è registrato al topic
- un messaggio può essere ricevuto da zero, uno o più (molti) consumatori





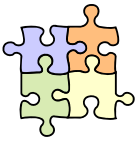
Argomenti e registrazioni durature

- **Argomento (*topic*) – *publisher-subscriber***
 - in generale, dunque, con gli argomenti c'è una dipendenza temporale nell'invio/ricezione di messaggi tramite un topic
 - questa dipendenza può essere rilassata tramite l'uso di "registrazioni durature" (durable subscription) da parte dei client consumatori
 - in questo caso, un consumatore può ricevere anche i messaggi che sono stati inviati quando lui non era attivo



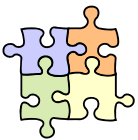
Comunicazione point-to-point

- La comunicazione ***point-to-point***
 - basata sui concetti di ***coda di messaggi (queue)***, ***sender*** e ***receiver***
 - ciascun messaggio è indirizzato da un sender a 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 ci sono più receiver registrati/interessati a una coda
 - non ci sono dipendenze temporali tra sender e receiver



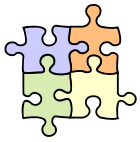
Comunicazione publisher/subscriber

- Nella comunicazione *publisher/subscriber*
 - i client publisher indirizzano i loro messaggi a un *topic* (*argomento*)
 - i client subscriber di messaggi si registrano dinamicamente ai topic di interesse
 - quando un publisher invia un messaggio a 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 può 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
 - questa dipendenza può essere rilassata



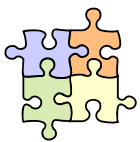
Discussione

- 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
 - attenzione al caso di zero consumatori
 - i messaggi possono essere effettivamente persi
 - se non è accettabile perdere messaggi, è necessario usare una configurazione dei componenti – ad esempio, un opportuno script per la creazione e l'avvio dei componenti produttori e consumatori – che garantisca che, in ogni momento in cui un publisher è attivo, ci sia almeno uno (o più) subscriber attivi registrati al topic

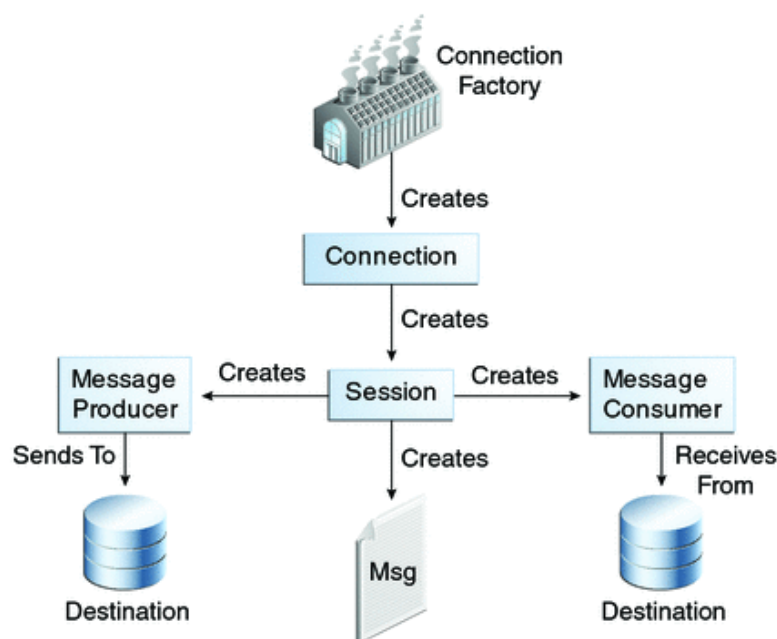


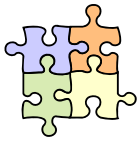
- API di JMS (1)

- 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 generalizzano questi concetti
 - ad es., **Destination**, **Connection**, **Session**, **MessageProducer** e **MessageConsumer**
- Inoltre, similmente a quando avviene con JDBC, il programmatore deve far riferimento solo a interfacce definite da JMS
 - non serve conoscere nessuna implementazione delle interfacce



API di JMS (2)





API di JMS (3)

□ **Destination**

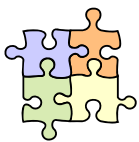
- rappresenta una destinazione (coda o topic) su un provider

□ **Connection**

- rappresenta una connessione virtuale tra un client e un provider JMS – è anche una factory di sessioni – *in qualche modo analoga ad una connessione JDBC*

□ **Session**

- è un contesto single-threaded per la produzione e il consumo di messaggi
 - single-threaded – come conseguenza, nell'ambito di una sessione, i messaggi sono serializzati, ovvero inviati e/o consumati uno alla volta, in modo sequenziale
- fornisce un contesto per l'elaborazione transazionale dei messaggi – *in qualche modo analoga ad una transazione JDBC*



API di JMS (4)

□ **MessageProducer**

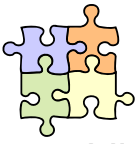
- un oggetto intermediario, di supporto, per inviare messaggi a una certa destinazione – per un client JMS, incapsula una destinazione su cui inviare messaggi
- non è il produttore “logico” del messaggio – ma è un oggetto di supporto necessario al produttore “logico”

□ **MessageConsumer**

- un oggetto intermediario, di supporto, per ricevere messaggi da una certa destinazione – per un client JMS, incapsula una destinazione da cui ricevere messaggi
- non è il consumatore “logico” del messaggio – ma è un oggetto di supporto necessario al consumatore “logico”

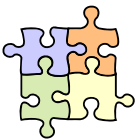
□ **MessageListener**

- interfaccia per la ricezione asincrona di messaggi



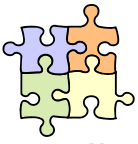
Invio di messaggi

- L'invio di un messaggio da parte di un client produttore richiede l'esecuzione di diverse attività
 - usare la `connection factory` per ottenere una connessione
 - `connection = connectionFactory.getConnection();`
 - usare la connessione per ottenere una sessione
 - `session = connection.createSession(...);`
 - usare la sessione per creare un message producer per una certa destinazione
 - `messageProducer = sessione.createProducer(destination);`
 - usare la sessione per creare un messaggio
 - `message = session.createTextMessage().setText(...);`
 - usare il message producer per inviare il messaggio
 - `messageProducer.send(message);`
 - infine, chiudere la connessione
 - `connection.close();`



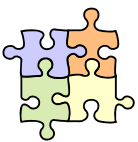
Ricezione (sincrona) di messaggi

- In modo analogo, anche la ricezione di un messaggio da parte di un client consumatore richiede l'esecuzione di diverse attività
 - ottenere una connessione e una sessione
 - `connection = connectionFactory.getConnection();`
 - `session = connection.createSession(...);`
 - usare la sessione per creare un message consumer per una certa destinazione
 - `messageConsumer = sessione.createConsumer(destination);`
 - abilitare la ricezione di messaggi
 - `connection.start();`
 - usare il message consumer per ricevere un messaggio
 - `message = messageConsumer.receive();`
 - arrestare la ricezione di messaggi e chiudere la connessione
 - `connection.stop(); connection.close();`



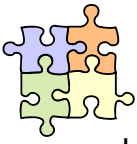
Consumo di messaggi

- Il consumo di messaggi può avvenire secondo due modalità
 - **consumo sincrono** – quello mostrato nell'esempio precedente – un consumatore legge i messaggi mediante un'operazione **receive** bloccante – e dunque in modo “sincrono”
 - **consumo asincrono** – in alternativa, un client consumatore può creare (e registrare) un consumatore che implementa un'interfaccia **message listener**
 - deve implementare un metodo **onMessage** – che dice che cosa bisogna fare quando viene ricevuto un messaggio
 - il client consumatore è un componente applicativo Java EE, e vive in un contenitore Java EE
 - il contenitore Java EE, quando rileva un messaggio in una certa destinazione, seleziona uno dei consumatori interessati a ricevere messaggi da quella destinazione – e gli comunica il messaggio invocandone il metodo **onMessage** – in modo “asincrono” rispetto al consumatore



Esempi e configurazione

- Negli esempi che seguono si assume che
 - sia stato installato e configurato un provider JMS
 - ad esempio, Oracle GlassFish Server v3.1.1
 - sul provider JMS siano stati definiti e configurati degli opportuni oggetti amministrati
 - una coda fisica `jms_asw840_Queue` e una coda logica `jms/asw840/Queue`
 - un argomento (topic) fisico e il corrispondente argomento (topic) logico (`jms/asw840/Topic`)
 - una connection factory `jms/asw840/ConnectionFactory` – specifica caratteristiche della modalità d'accesso alla destinazione, ad es., l'affidabilità
 - le applicazioni mostrate siano compilate e eseguite come **application client** Java EE mediante un IDE opportuno
 - in particolare, Eclipse o NetBeans



Esempio - produttore di messaggi (1)

```
package asw.asw840.simpleproducer;

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

public class Main {

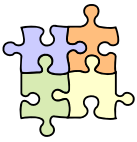
    @Resource(lookup = "jms/asw840/Queue")
    private static Queue queue;
    @Resource(lookup = "jms/asw840/Topic")
    private static Topic topic;
    @Resource(lookup = "jms/asw840/ConnectionFactory")
    private static ConnectionFactory connectionFactory;

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



Annotazioni e iniezione di risorse

- Nell'esempio, l'annotazione `@Resource` consente di "iniettare" il riferimento a una risorsa in una variabile
 - quando il compilatore incontra un'annotazione, la riporta, come metadato, nel bytecode – il compilatore non interpreta le annotazioni
 - le annotazioni vengono invece normalmente prese in considerazione dagli strumenti di sviluppo (ad es., JUnit) e/o dall'ambiente di esecuzione – nel nostro caso, dall'ambiente in cui viene eseguito un application client ("appclient")
- In particolare, 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



Iniezione di risorse - in Java EE

- 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;
```

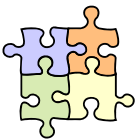
Java EE 5 o 6

```
@Resource(lookup = "jms/asw840/Queue")  
private static Queue queue;
```

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

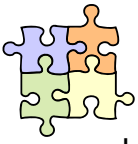
Java EE 1.4

```
String queueName = "jms/asw840/Queue";  
Context jndiContext = new InitialContext();  
Queue queue = (javax.jms.Queue) jndiContext.lookup(queueName);
```



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();  
}
```



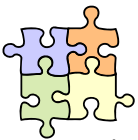
Esempio - produttore di messaggi (3)

```
package asw.asw840.simpleproducer;
import javax.jms.*;

public class SimpleProducer {

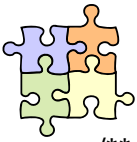
    /* nome di questo producer */
    private String name;
    /* destinazione di questo producer */
    private Destination destination;
    /* connection factory di questo producer */
    private ConnectionFactory connectionFactory;

    /* connessione jms */
    private Connection connection = null;
    /* sessione jms */
    private Session session = null;
    /* per l'invio di messaggi alla destinazione */
    private MessageProducer messageProducer = null;
    ...
}
```



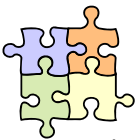
Esempio - produttore di messaggi (4)

```
/** 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;
}
```



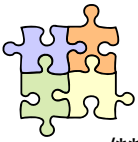
Esempio - produttore di messaggi (5)

```
/** Si connette alla destinazione JMS.  
 * Crea anche una sessione, e un message producer per la destinazione. */  
public void connect() {  
    try {  
        connection = connectionFactory.createConnection();  
        session = connection.createSession( false, // non transazionale  
                                           Session.AUTO_ACKNOWLEDGE);  
        messageProducer = session.createProducer(destination);  
    } catch (Exception e) {  
        System.out.println("Connection problem: " + e.toString());  
        disconnect();  
    }  
}
```



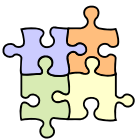
Esempio - produttore di messaggi (6)

```
/** Si disconnette dalla destinazione JMS. */  
public void disconnect() {  
    if (connection != null) {  
        try {  
            connection.close();  
            connection = null;  
        } catch (JMSEException 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 {
        TextMessage message = session.createTextMessage();
        message.setText(text);
        /* ora invia il messaggio */
        messageProducer.send(message);
    } catch (JMSEException e) {
        System.out.println("Error sending message: " + e.toString());
    }
}
```



Esempio - consumatore sincrono (1)

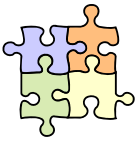
```
package asw.asw840.simplesynchconsumer;

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

public class Main {

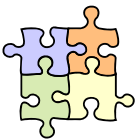
    come prima {
        @Resource(lookup = "jms/asw840/Queue")
        private static Queue queue;
        @Resource(lookup = "jms/asw840/Topic")
        private static Topic topic;
        @Resource(lookup = "jms/asw840/ConnectionFactory")
        private static ConnectionFactory connectionFactory;

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



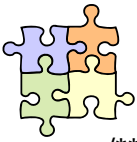
Esempio - consumatore sincrono (2)

```
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();  
        ... fa qualcosa con il messaggio message ricevuto ...  
    }  
  
    /* termina la ricezione dei messaggi e chiude la connessione */  
    simpleConsumer.stop();  
    simpleConsumer.disconnect();  
}
```



Esempio - consumatore sincrono (3)

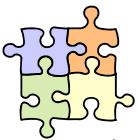
```
package asw.asw840.simplesynchconsumer;  
import javax.jms.*;  
  
public class SimpleSynchConsumer {  
  
    /* nome di questo consumer */  
    private String name;  
    /* destinazione di questo consumer */  
    private Destination destination;  
    /* connection factory di questo consumer */  
    private ConnectionFactory connectionFactory;  
  
    /* connessione jms */  
    private Connection connection = null;  
    /* sessione jms */  
    private Session session = null;  
    /* per la ricezione dei messaggi dalla destinazione */  
    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;
}
...

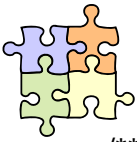
```



Esempio - consumatore sincrono (5)

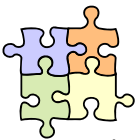
```
/** Si connette alla destinazione JMS.
 * Crea anche una sessione, e un message consumer per la destinazione. */
public void connect() {
    try {
        connection = connectionFactory.createConnection();
        session = connection.createSession( false, // non transazionale
                                           Session.AUTO_ACKNOWLEDGE);
        messageConsumer = session.createConsumer(destination);
    } catch (Exception e) {
        System.out.println("Connection problem: " + e.toString());
        disconnect();
    }
}
... disconnect, come per il produttore ...
...

```



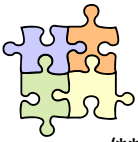
Esempio - consumatore sincrono (6)

```
/** Avvia la ricezione dei messaggi */
public void start() {
    try {
        /* avvia la consegna di messaggi per la connessione */
        connection.start();
    } catch (JMSEException e) {
        System.out.println("Error starting message reception: " + e.toString());
        System.exit(1);
    }
}
```



Esempio - consumatore sincrono (7)

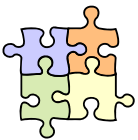
```
/** Arresta la ricezione dei messaggi */
public void stop() {
    try {
        /* arresta la consegna di messaggi per la connessione */
        connection.stop();
    } catch (JMSEException e) {
        System.out.println("Error stopping message reception: " + e.toString());
        System.exit(1);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



Esempio - consumatore sincrono (8)

```
/** 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 (JMSEException e) {
        System.out.println("Error receiving message: " + e.toString());
        System.exit(1);
    }
    return null;
}
```



Esempio - consumatore asincrono (1)

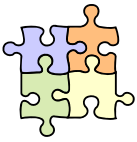
```
package asw.asw840.simpleasynchconsumer;

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

public class Main {

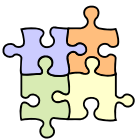
    come prima {
        @Resource(lookup = "jms/asw840/Queue")
        private static Queue queue;
        @Resource(lookup = "jms/asw840/Topic")
        private static Topic topic;
        @Resource(lookup = "jms/asw840/ConnectionFactory")
        private static ConnectionFactory connectionFactory;

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



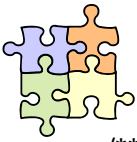
Esempio - consumatore asincrono (2)

```
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 – li riceve e li elabora */  
    simpleConsumer.receiveMessages();  
  
    /* chiude la connessione */  
    simpleConsumer.disconnect();  
}
```



Esempio - consumatore asincrono (3)

```
package asw.asw840.simpleasynchconsumer;  
  
import javax.jms.*;  
  
public class SimpleAsynchConsumer {  
    /* 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 dalla destinazione */  
    private MessageConsumer messageConsumer = null;  
  
    ...  
}
```

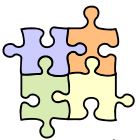


Esempio - consumatore asincrono (4)

```
/** 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;  
}
```

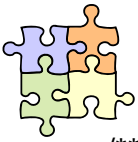
... connect e disconnect, come per il consumatore sincrono ...

...



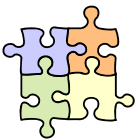
Esempio - consumatore asincrono (5)

```
/** Riceve messaggi dalla destinazione */  
public void receiveMessages() {  
    try {  
        /* 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 (JMSEException e) { ... }  
}
```



Esempio - consumatore asincrono (6)

```
/** Definisce la logica applicativa associata alla ricezione di un messaggio.
 * Si tratta del metodo (di callback) chiamato dal TextListener. */
public void messageReceived(String message) {
    ... fa qualcosa con il messaggio message ricevuto ...
}
```



Esempio - consumatore asincrono (7)

```
package asw.asw840.simpleasynchconsumer;

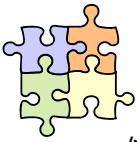
import javax.jms.*;

public class TextListener implements MessageListener {

    /** consumer di questo TextListener */
    SimpleAsynchConsumer consumer;

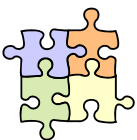
    /** Crea un nuovo TextListener per il consumer c. */
    public TextListener(SimpleAsynchConsumer c) {
        this.consumer = c;
    }

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



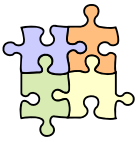
Esempio - consumatore asincrono (8)

```
/** Riceve un messaggio. Delega la sua gestione al metodo di callback. */  
public void onMessage(Message m) {  
    if (m instanceof TextMessage) {  
        TextMessage message = (TextMessage) m;  
        try {  
            consumer.messageReceived(message.getText());  
        } catch (JMSEException e) {  
            System.out.println("TextListener.onMessage(): " + e.toString());  
        }  
    }  
}
```



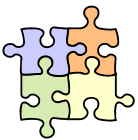
Il metodo onMessage()

- Chi invoca il metodo onMessage? Quando?
 - si consideri ad esempio una certa destinazione (coda) D, un produttore P per D e due consumatori asincroni C1 e C2 (potrebbero essere due istanze di una stessa classe o addirittura di due classi diverse) per D che
 - hanno dichiarato di essere consumatori per D
 - hanno avviato (start) la ricezione di messaggi da D
 - che succede quando P invia un messaggio M su D?
 - in prima battuta, M non viene ricevuto né da C1 né da C2
 - piuttosto, l'invio del messaggio M su D viene preso in carico dall'application server (AS)
 - è l'AS che sa quali consumatori sono abbonati a una certa destinazione – ed è lui che decide a chi (C1 oppure C2) consegnare il messaggio M – e la consegna avviene invocando il metodo onMessage di uno di questi due oggetti



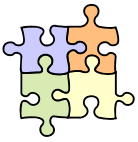
Osservazioni

- 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
 - miglior separazione tra componenti e connettori
 - comportamento del consumatore basato sul contenuto dei messaggi
 - la ragion d'essere del messaging
 - uso di un meccanismo per consentire la terminazione dei consumatori
 - ad esempio, un messaggio che indica la “fine della sessione”



- Alcuni esperimenti con le code

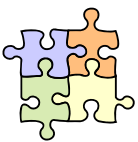
- Esperimento A con una coda
 - avvio il consumatore, poi avvio il produttore che invia N messaggi
 - conseguenze
 - il produttore invia N messaggi
 - il consumatore riceve N messaggi



Alcuni esperimenti con le code

□ Esperimento B con una coda

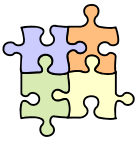
- il consumatore non è inizialmente attivo, il produttore invia N messaggi
 - il produttore invia N messaggi, e termina
- poi viene avviato il consumatore
- conseguenze
 - il consumatore riceve N messaggi – a meno che sia passato un tempo tale da far “scadere” i messaggi inviati



Alcuni esperimenti con le code

□ Esperimento C con una coda

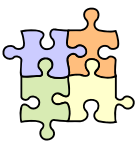
- avvio il consumatore, e due produttori sulla stessa coda
- conseguenze
 - un produttore invia N messaggi
 - un produttore invia M messaggi
 - il consumatore riceve N+M messaggi



Alcuni esperimenti con le code

□ Esperimento D con una coda

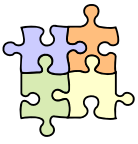
- avvio due consumatori sulla stessa coda, poi il produttore invia N messaggi
- conseguenze
 - il produttore invia N messaggi
 - un consumatore riceve X messaggi
 - l'altro consumatore riceve gli altri N-X messaggi



Alcuni esperimenti con le code

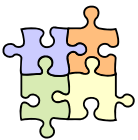
□ Esperimento E con una coda

- avvio due consumatori sulla stessa coda, poi due produttori sulla stessa coda
- conseguenze
 - un produttore invia N messaggi
 - un produttore invia M messaggi
 - un consumatore riceve X messaggi
 - l'altro consumatore riceve gli altri N+M-X messaggi



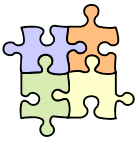
Alcuni esperimenti con gli argomenti

- 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



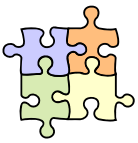
Alcuni esperimenti con gli argomenti

- 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



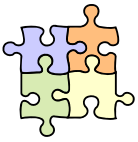
Alcuni esperimenti con gli argomenti

- 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
 - ciascuno dei consumatori riceve N messaggi

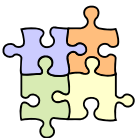


- Ulteriori considerazioni

- Si vuole realizzare un'applicazione di tipo pipes and filters, in cui
 - un produttore (P) genera un flusso di messaggi
 - un primo filtro (F1) riceve questo flusso di messaggi – li trasforma, e genera un secondo flusso di messaggi
 - un secondo filtro (F2) riceve questo secondo flusso di messaggi – li trasforma, e genera un terzo flusso di messaggi
 - un consumatore (C) consuma questo terzo flusso di messaggi

 - come fare in modo che ciascun componente consumi i messaggi giusti – ad esempio che il filtro F2 consumi solo i messaggi generati dal filtro F1 – ma non quelli generati dal produttore P?

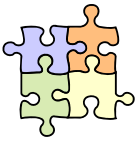
 - evidentemente, è necessario usare più destinazioni intermedie
 - una tra P e F1, una tra F1 e F2, una tra F2 e C



Ulteriori considerazioni

- 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 (a runtime) delle destinazioni temporanee
 - un messaggio di richiesta può contenere un campo ReplyTo – specifica dove inviare la risposta a quel messaggio



Informazioni associate ai messaggi

- Alcune informazioni utili che sono (o possono essere) associate a un messaggio
 - 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



- Discussione

- Molto bello, ma...
 - qual è il modo corretto di usare il messaging in pratica?
 - quali i campi di applicazione reali?
 - ...
- Questi aspetti sono di interesse metodologico – e saranno trattati come tali nel seguito del corso
 - l'applicazione della tecnologia del messaging può essere guidata, in generale, dallo stile architetturale pipes and filters
 - un campo di applicazione significativo del messaging è l'integrazione di applicazioni
 - in questo contesto, l'applicazione della tecnologia del messaging è guidata dallo stile architetturale “messaging” – insieme ad altri pattern di supporto a questo stile