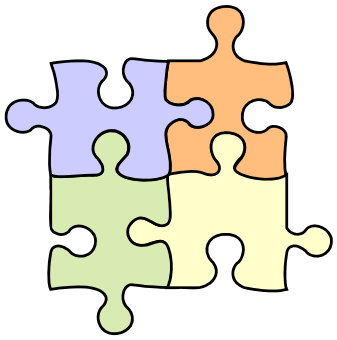
Luca Cabibbo



Architetture Software

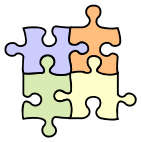
Messaging (middleware)

Dispensa ASW 840

ottobre 2014

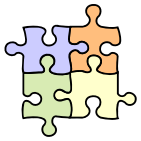
Quando sei di fronte a un'interfaccia particolarmente difficile, prova a cambiare la sua caratterizzazione.

Eberhardt Rechtin



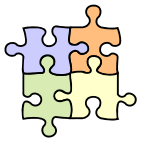
- Fonti

- [EIP] Hohpe, Woolf, Enterprise Integration Patterns, 2004
  - *http://www.enterpriseintegrationpatterns.com/*
  - *http://eaipatterns.com/*
- The Java EE 7 Tutorial
  - *http://docs.oracle.com/javaee/7/tutorial/doc/*
  - Chapter 45, Java Message Service Concepts
  - Chapter 46, Java Message Service Examples

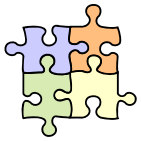


- Obiettivi e argomenti

- Obiettivi
  - presentare il messaging – qui inteso come tecnologia di middleware
  - come studio di caso, introdurre Java Message Service (JMS)
- Argomenti
  - messaging
  - Java Message Service (JMS)

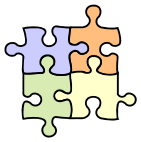


- Wordle



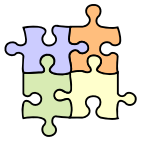
* Messaging

- Il **messaging** è un paradigma/una tecnologia di comunicazione per sistemi distribuiti
  - i componenti o le applicazione comunicano mediante lo scambio di messaggi
    - un *componente* (nel ruolo di *produttore*) può inviare *messaggi* a un altro *componente* (nel ruolo di *consumatore*) mediante un *canale di comunicazione* intermedio
  - questa modalità di comunicazione è supportata da opportuni strumenti di middleware



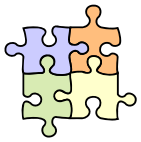
Confronto tra messaging e RPC/RMI

- Il messaging è un paradigma di comunicazione significativamente diverso da quello richiesta/risposta su cui sono basati RPC e RMI
  - con RPC/RMI
    - i componenti che partecipano a un'interazione sono il client e il server per un servizio
    - il client chiede al server l'erogazione del servizio – il server esegue l'operazione richiesta e fornisce una risposta al client
    - l'interazione è diretta e sincrona
  - nel messaging, invece
    - i partecipanti di una specifica interazione sono il produttore e il consumatore di un messaggio – di solito operano da "pari"
    - la comunicazione è iniziata dal produttore – il consumatore elabora il messaggio ricevuto – e di solito non fornisce una risposta al produttore
    - l'interazione è indiretta ed asincrona

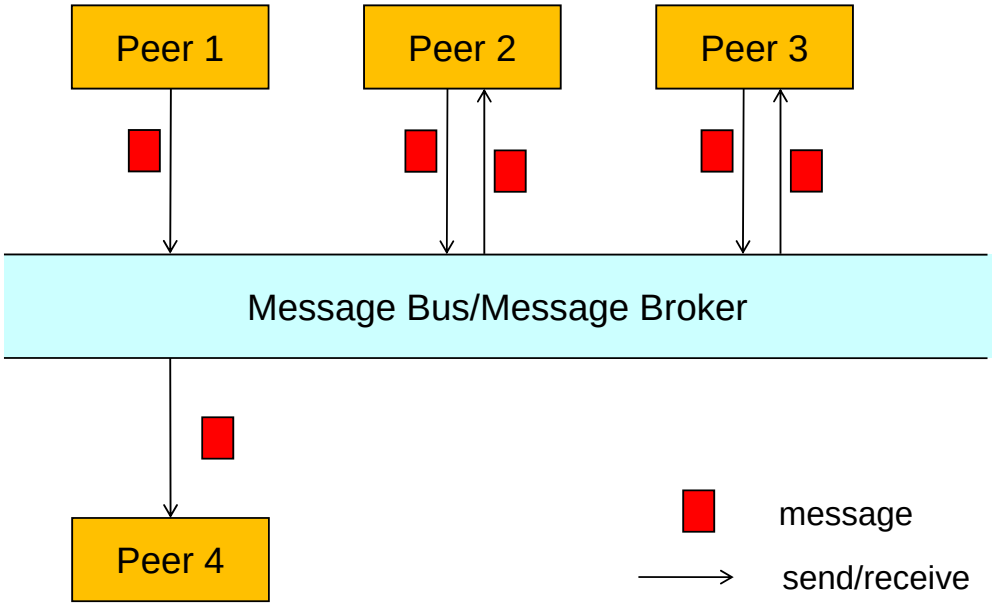


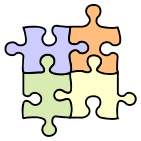
Caratteristiche del messaging

- Nel messaging, la comunicazione viene avviata dal componente (*produttore*) che produce il messaggio
  - e non dal componente (*consumatore*) che poi utilizzerà ed elaborerà le informazioni contenute nel messaggio
- Nel messaging, 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*
  - il consumatore poi leggerà messaggi da questo canale
- Nel messaging, la comunicazione è *asincrona*
  - il consumatore può leggere un messaggio anche in un momento diverso da quando il messaggio è stato inviato
  - ovvero, il consumo non avviene necessariamente nello stesso momento in cui il messaggio è stato prodotto e inviato



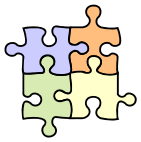
Un sistema di messaging



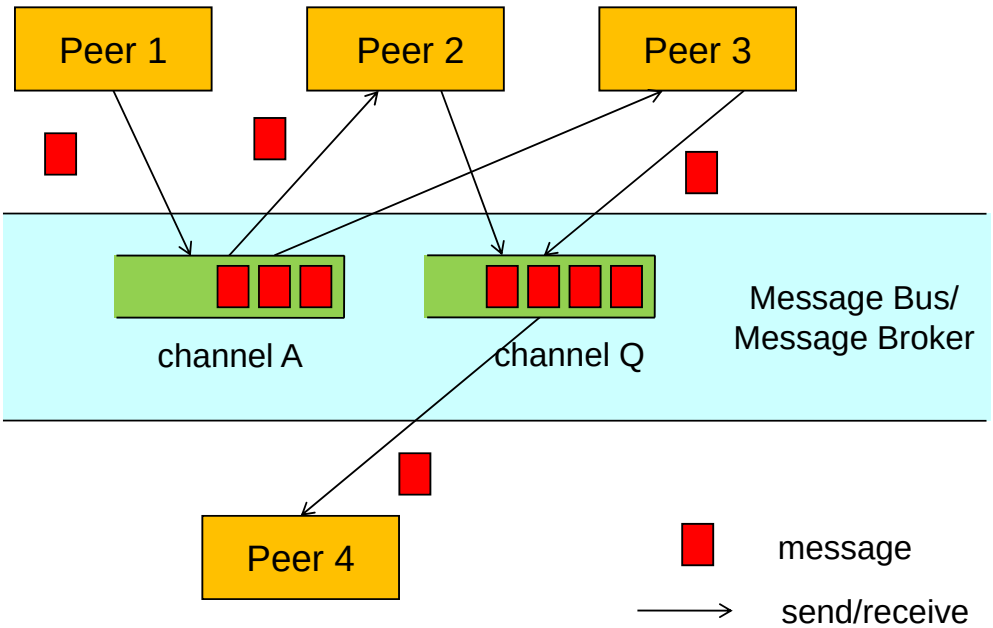


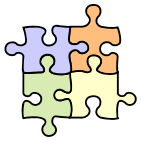
Destinazioni intermedie

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



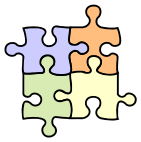
Un sistema di messaging



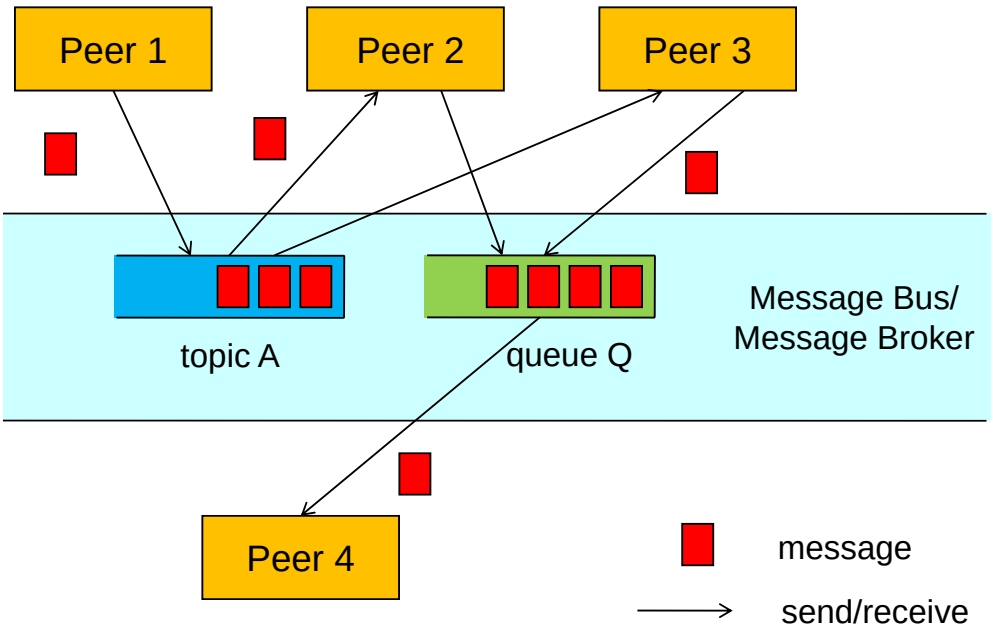


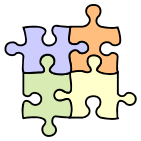
Destinazioni intermedie

- Esistono due tipi principali di destinazioni intermedie – che differiscono nella semantica dell'interazione
  - *queue – coda*
    - un messaggio inviato a una coda sarà consumato da uno e un solo consumatore
    - è dunque un canale di comunicazione "a-uno"
    - attenzione, questo non vuol dire che a ogni coda può essere associato un solo consumatore
  - *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-suscriber



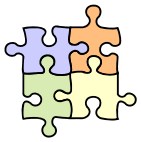
Un sistema di messaging





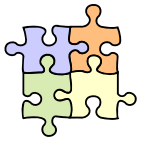
Produttori, consumatori, destinazioni

- In generale, in un'applicazione di messaging ci sono
  - una molteplicità di destinazioni intermedie – code e topic – ciascuna destinazione veicola 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 a 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
    - questo è probabilmente anche il caso più comune



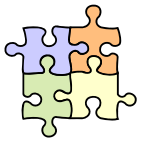
Invocazione implicita

- 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 *dal consumatore* 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"

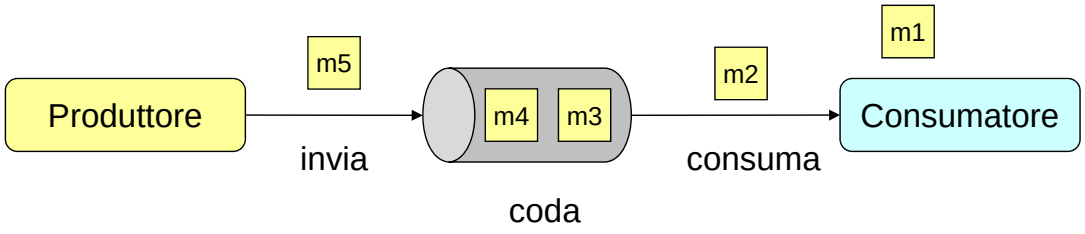


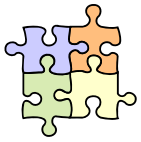
Invocazione implicita

- 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
  - ciascun consumatore è in grado di gestire richieste (di un certo tipo)
  - i consumatori, quando sono disponibili ad accettare richieste, leggono 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 questo caso, crea un nuovo messaggio e lo invia a un'altra destinazione, affinché l'elaborazione prosegua

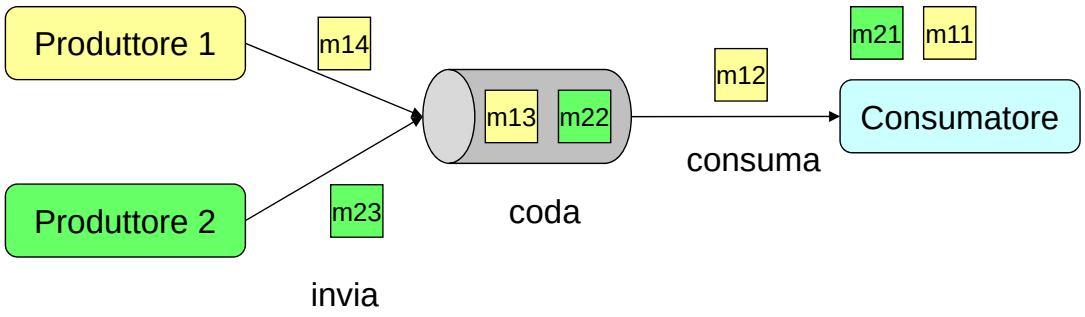


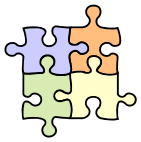
Messaging – code



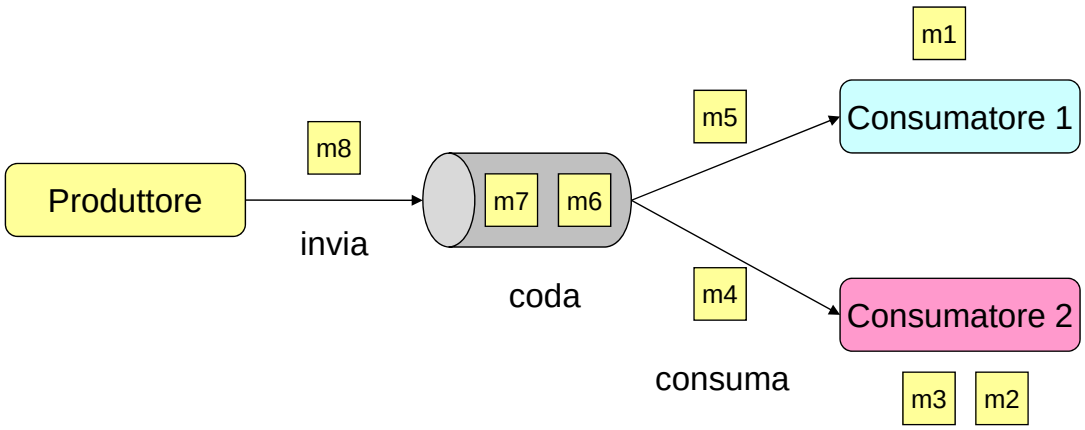


Code: più produttori



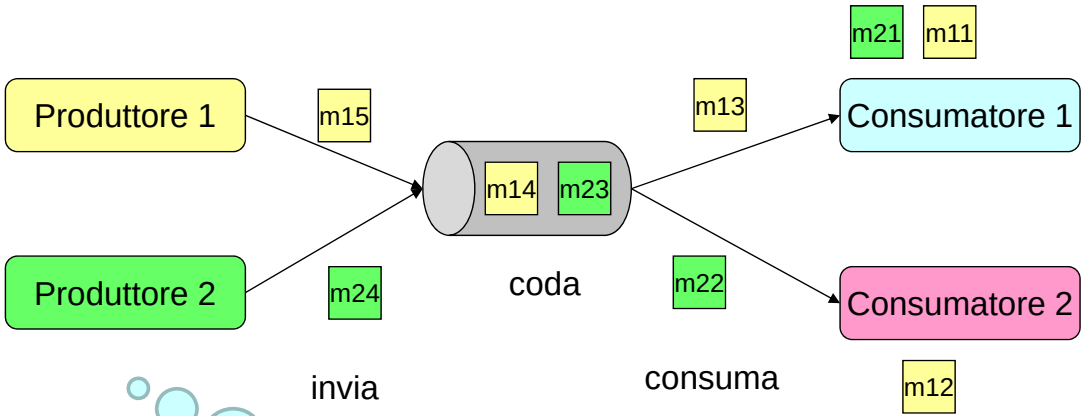


Code: più consumatori



In questo caso, ciascun messaggio viene consumato da un solo consumatore.

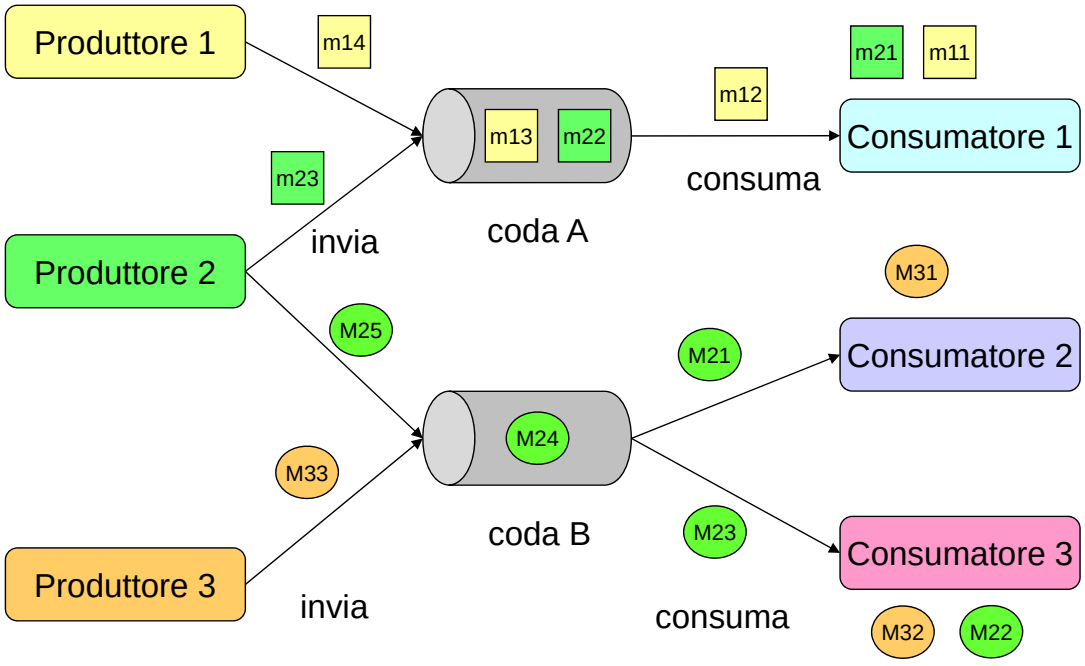
Code: più produttori/più consumatori

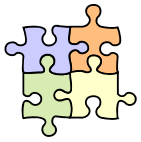


Anche in questo caso, ciascun messaggio viene consumato da un solo consumatore.

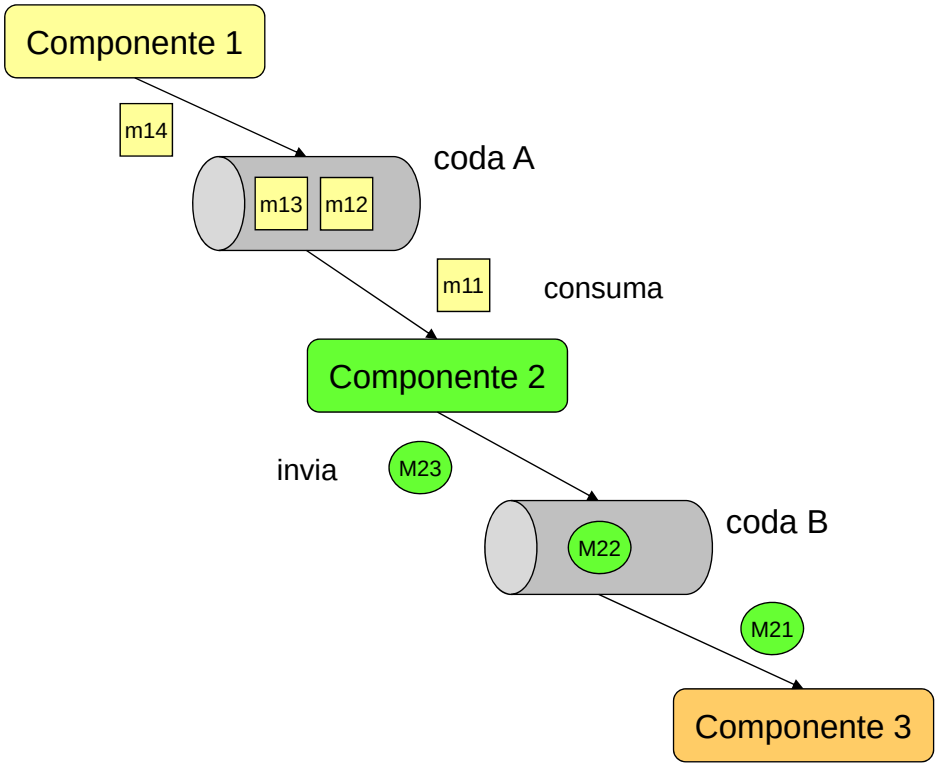
Messaggi di uno stesso produttore possono essere consumati da consumatori diversi.

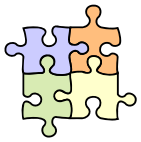
Code: più destinazioni (più tipi di messaggi)



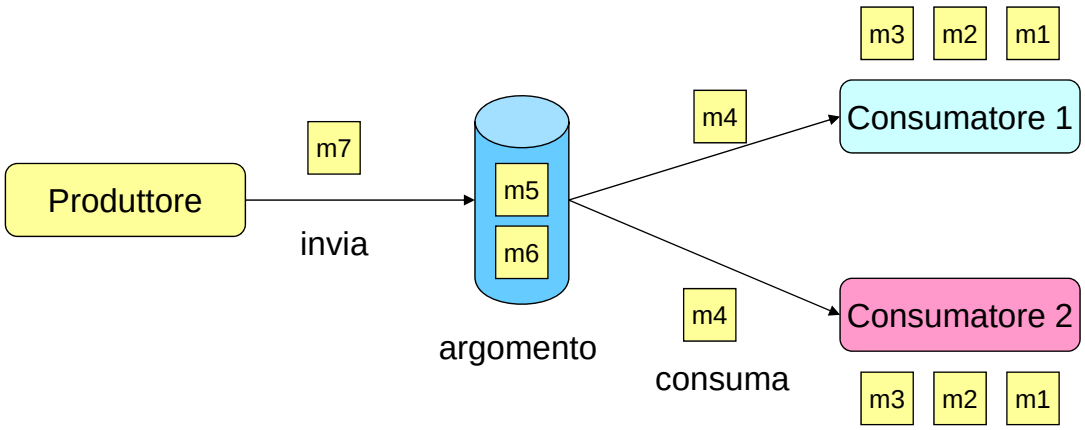


Code: componenti consumatori/produttori

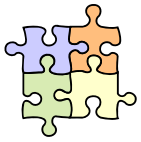




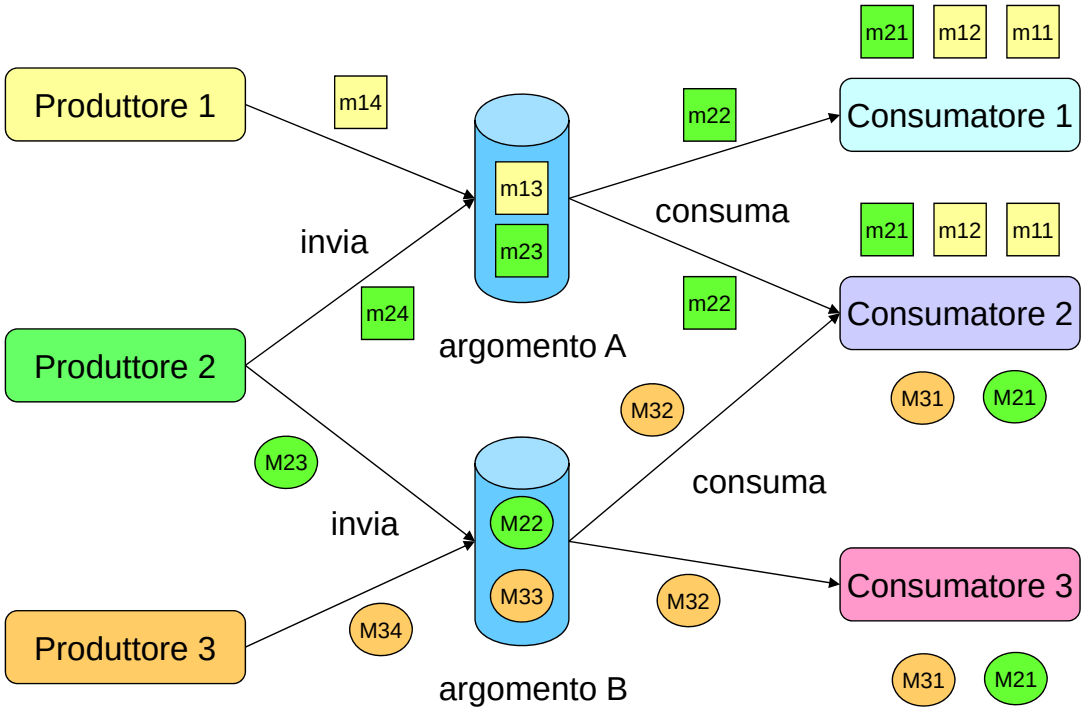
Messaging: argomenti

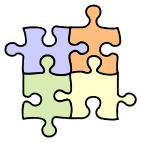


In questo caso, i messaggi possono essere consumati da più consumatori.



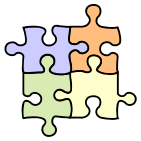
Messaging: argomenti





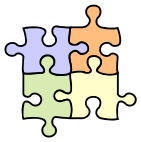
Produttori e consumatori

- Il messaging è una modalità di comunicazione tra “pari” (*peer*)
  - in linea di principio, un componente può inviare messaggi a e ricevere messaggi da ogni altro componente
    - attenzione, 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 il (o un) *consumatore* del messaggio
  - dunque, la caratteristica di essere “produttore” o “consumatore” non è in generale propria di un componente – ma è specifica del ruolo svolto da un componente nello scambio di un particolare messaggio



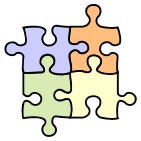
Produttori e consumatori

- Per comunicare, una coppia di componenti (produttore e consumatore) devono essere d'accordo
  - sul formato dei messaggi da scambiare, e
  - su quale destinazione intermedia usare
- 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



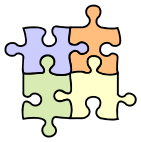
Comunicazione asincrona

- 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'indirezione asincrona 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



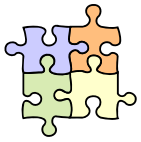
Analogie

- 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, SMS, oppure su un social network
  - 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



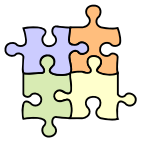
Messaging e accoppiamento debole

- 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, e all'interfaccia procedurale dei componenti che devono comunicare
    - non devono nemmeno essere attivi contemporaneamente
  - l'accoppiamento richiesto per comunicare è *più debole* che non nella comunicazione di tipo RPC/RMI



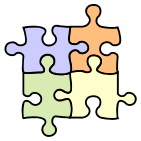
Messaging e affidabilità

- Nei sistemi di messaging, la consegna dei messaggi può essere caratterizzata da diversi livelli di *affidabilità* – selezionabile 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
  - è anche possibile una consegna “persistente” dei messaggi – in cui viene garantito che un messaggio (inviato a una coda) sia consegnato (o consegnato con successo) una e una sola volta
  - un componente può anche gestire messaggi in modo transazionale – in cui viene garantita non solo la ricezione ma anche l’elaborazione di un messaggio o anche di un gruppo di messaggi
  - in generale, il livello di affidabilità può essere (e va) configurato in modo dipendente anche dal contesto applicativo



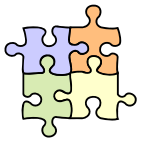
Messaging, prestazioni e scalabilità

- Il messaging è considerato anche una tecnologia che abilita *prestazioni* e *scalabilità*
  - nel messaging, i messaggi vengono consegnati in modo asincrono – in ogni caso, il middleware cerca di consegnarli appena possibile, con una latenza bassa
  - l’uso di più istanze dei componenti consumatori (*Maintain multiple copies of computations*) favorisce prestazioni e scalabilità
  - in effetti, quest’ultima scelta è efficace se i componenti sono stateless – questo favorisce ulteriormente la scalabilità
  - in pratica, il messaging può semplificare in modo significativo la realizzazione di applicazioni scalabili sul cloud
  - attenzione, il messaging è una tecnologia abilitante per queste qualità – tuttavia, *da solo* non può garantirle



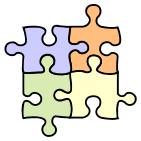
Quando usare un sistema di messaging?

- 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 facilita la sostituzione dei componenti produttori e consumatori
  - come caso estremo, se i componenti che devono comunicare sono stati sviluppati indipendentemente l'uno dall'altro



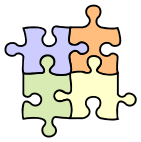
Messaging server-centrico e distribuito

- L'implementazione di alcuni sistemi di messaging è *server-centrica*
  - nel senso che le destinazioni sono gestite da un application server centralizzato
  - per scrivere/leggere sulle/dalle destinazioni, l'application server deve essere attivo
- L'implementazione di altri sistemi di messaging è invece *distribuita*
  - le destinazioni sono gestite in modo distribuito su un insieme di host – su ciascun host è in esecuzione un opportuno demone



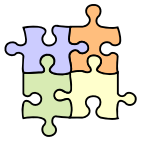
* Java Message Service (JMS)

- *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



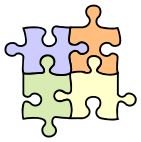
What is messaging?

- **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



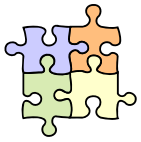
What is the JMS API?

- 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
- The JMS API **enables communication** that is not only **loosely coupled** but also
  - **asynchronous**: a receiving client does not have to receive messages at the same time the sending client sends them; the sending client can send them and go on to other tasks; the receiving client can receive them much later
  - **reliable**: a messaging provider that implements 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



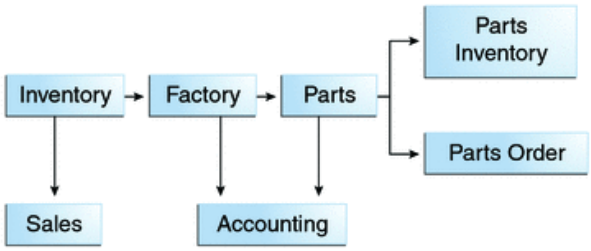
When can you use the JMS API?

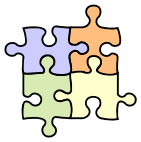
- An enterprise application provider is likely to choose a messaging API over a tightly coupled API, such as remote procedure call (RPC), under the following circumstances
  - the provider wants the components not to depend on information about other components' interfaces, so that components can be 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?

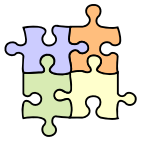
- For example, components of an enterprise application for an automobile manufacturer can use the JMS API in situations like these
  - the inventory component can send a message to the factory component when the inventory level for a product goes below a certain level so that the factory can make more 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





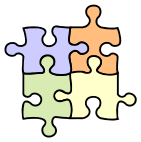
How does the JMS API work with the Java EE Platform?

- The JMS API is an integral part of the Java EE platform, and application developers can use messaging with Java EE components
  - application clients, Enterprise JavaBeans (EJB) components, and web components can send or synchronously receive a JMS message; application clients can in addition set a message listener that allows JMS messages to be delivered to it asynchronously by being notified when a message is available
  - message-driven beans, which are a kind of enterprise bean, enable the asynchronous consumption of messages in the EJB container; an application server typically pools message-driven beans to implement concurrent processing of messages
  - message send and receive operations can participate in Java Transaction API (JTA) transactions, which allow JMS operations and database accesses to take place within a single transaction
- The JMS API enhances the other parts of the Java EE platform by simplifying enterprise development, allowing loosely coupled, reliable, asynchronous interactions among Java EE components and legacy systems capable of messaging
  - a developer can easily add new behavior to a Java EE application that has existing business events by adding a new message-driven bean to operate on specific business events
  - the Java EE platform, moreover, enhances the JMS API by providing support for JTA transactions and allowing for the concurrent consumption of messages



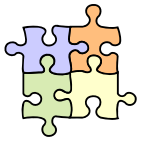
JMS – concetti di base (1)

- 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
    - gli AS che implementano la piattaforma Java EE possono comprendere un provider JMS – ad es., Glassfish
    - attenzione, esistono anche implementazioni "parziali" di Java EE – che potrebbero non comprendere un provider JMS
    - nel mondo .NET, il sistema operativo Windows può anche fungere da provider di servizi di messaging
  - la nozione di "provider" di un servizio esiste anche in altri contesti
    - ad esempio, un DBMS può essere considerato un provider di un "servizio di basi di dati"



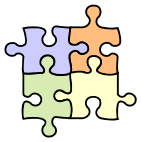
JMS – concetti di base (2)

- Le applicazioni basate su JMS fanno riferimento a un certo numero di concetti ed elementi
  - uno o più *client JMS* – i programmi o componenti, scritti in Java, che sono produttori e/o consumatori di messaggi
    - ad es., un qualunque componente applicativo Java EE (come un componente web o un EJB), oppure un application client Java EE, o anche un'applicazione Java SE
    - attenzione, JMS usa il termine "client" anche se, in effetti, questi componenti agiscono di solito come "peer"
  - i *messaggi* – sono oggetti che rappresentano informazioni scambiate dai client JMS

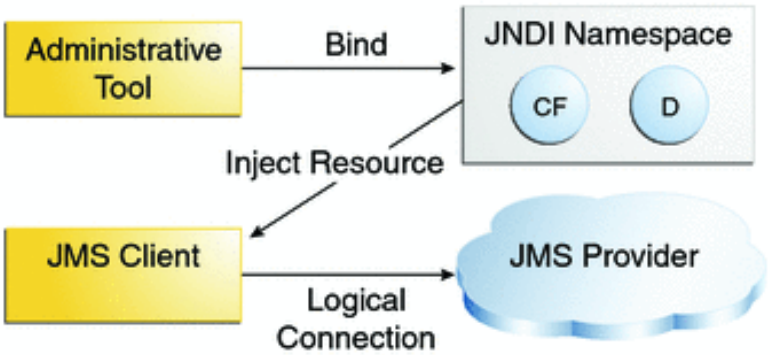


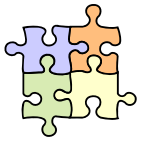
JMS – concetti di base (3)

- Le applicazioni basate su JMS fanno riferimento a un certo numero di concetti ed elementi
  - *oggetti amministrati* – sono oggetti JMS che vanno definiti/pre-configurati da un amministratore per essere usati dai client
    - in particolare, le *destinazioni* (code oppure argomenti) sono oggetti amministrati, così come le *factory per le connessioni*
    - l'insieme degli oggetti amministrati è analogo allo schema di una base di dati relazionale per un DBMS
    - attenzione – le caratteristiche e la semantica degli oggetti amministrati possono variare da provider a provider
  - *strumenti di amministrazione*
    - per creare/configurare le destinazioni e altri oggetti amministrati – ad es., la console di Glassfish
  - *un servizio di directory JNDI*
    - per l'accesso distribuito a risorse mediante nomi simbolici



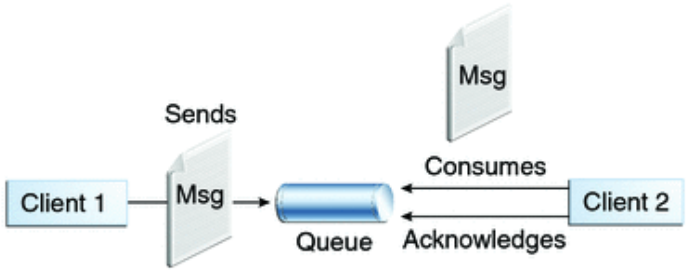
Architettura per JMS



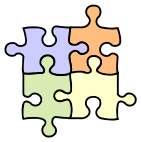


Destinazioni JMS – Code

- *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 inviato a una coda viene consumato da uno e un solo consumatore

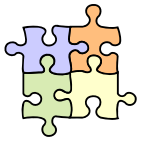


  - la coda conserva i messaggi che gli sono stati inviati fino a quando questi messaggi non sono stati consumati – oppure non sono "scaduti"



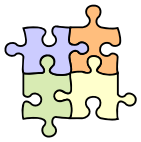
Destinazioni JMS – Topic

- *Argomento* (*topic*) – *publisher-subscriber*
  - un client consumatore si può registrare dinamicamente a diversi topic
  - un produttore può indirizzare messaggi a un topic specifico
  - un client consumatore riceve notifica dei messaggi inviati ai topic a cui è registrato
  - pertanto, ciascun messaggio può essere ricevuto da zero, uno o più (molti) consumatori



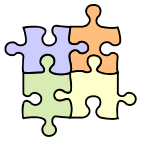
Destinazioni JMS – Topic

- *Argomento* (*topic*) – *publisher-subscriber*
  - usando una destinazione di tipo topic, il provider JMS si occupa di distribuire i messaggi prodotti dai "publisher" per quel topic a tutti i "subscriber" del topic
  - a differenza delle code, di solito un topic mantiene i messaggi solo per il tempo necessario per distribuirlo ai diversi abbonati – e poi li scarica
  - pertanto, un client abbonato riceverà normalmente i messaggi inviati al topic solo limitatamente al periodo di tempo in cui è registrato al topic ed attivo
    - ovvero, con i topic c'è normalmente una dipendenza temporale nell'invio/ricezione di messaggi tramite un topic
    - questa dipendenza può essere rilassata tramite l'uso di "abbonamenti duraturi" (durable subscription), descritti più avanti



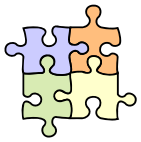
Comunicazione point-to-point

- Le due tipologie di destinazioni definiscono due diverse modalità di comunicazione
- Comunicazione *point-to-point*
  - basata sull'uso di una *coda di messaggi* (*queue*)
  - il produttore è chiamato *sender*, il consumatore *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



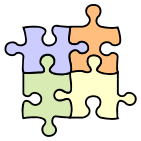
Comunicazione publisher/subscriber

- Comunicazione *publisher/subscriber*
  - basata sull'uso di un *argomento* (*topic*)
  - il produttore è chiamato *publisher*, il consumatore *subscriber*
  - i publisher indirizzano i loro messaggi a un *topic*
  - i subscriber si registrano dinamicamente ai topic di interesse
  - quando un publisher invia un messaggio a un topic, il provider distribuisce il messaggio 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
    - questa dipendenza può essere rilassata



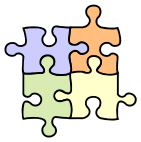
Discussione

- 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 – e spesso questo non è accettabile
    - per evitare questa situazione, è necessario definire e garantire un'opportuna configurazione dei componenti – ad esempio, mediante uno script per la creazione e l'avvio dei componenti produttori e consumatori – tale che, in ogni momento in cui un publisher è attivo, c'è almeno uno (o più) subscriber attivi registrati al topic

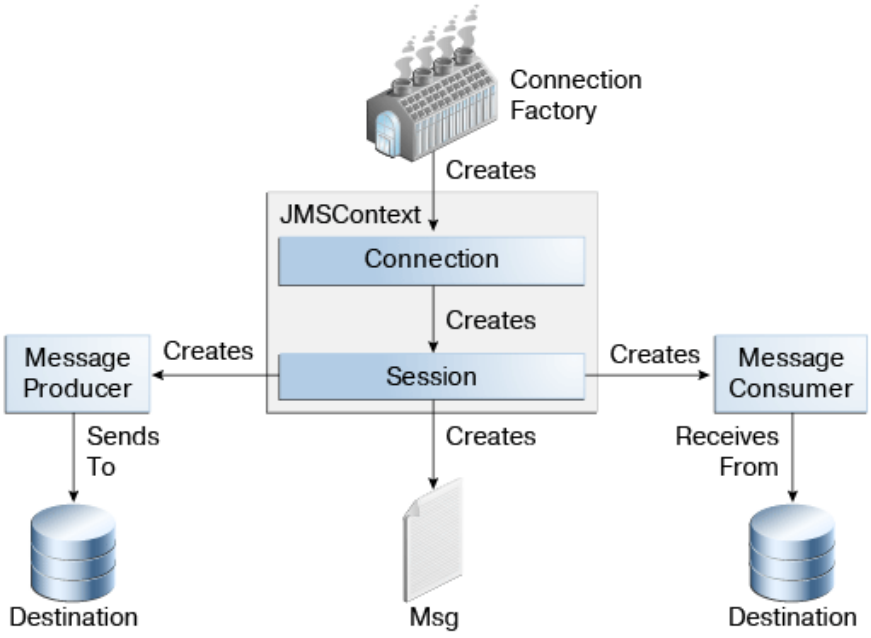


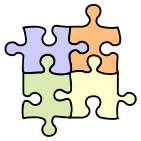
- API di JMS (1)

- Le API di JMS forniscono
  - 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***, ***JMSContext***, ***JMSProducer*** e ***JMSConsumer***
- 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



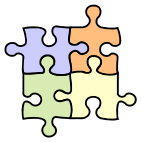
API di JMS (2)





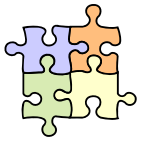
API di JMS (3)

- ***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 a una connessione JDBC*
- ***Session***
  - è un contesto single-threaded per la produzione e il consumo di messaggi
    - le sessioni sono single-threaded – vuol dire che, nell'ambito di ciascuna 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 a una transazione JDBC*



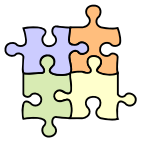
API di JMS (4)

- ***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 utile (anzi, necessario) a un client produttore di messaggi
- ***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 – è un oggetto di supporto utile/necessario a un client consumatore di messaggi
- ***MessageListener***
  - interfaccia per la ricezione asincrona di messaggi



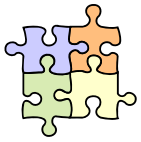
API di JMS (5)

- ***ConnectionFactory***
  - è l'oggetto usato dal client per ottenere una connessione con il provider
  - l'accesso alla connection factory avviene di solito mediante un'iniezione di dipendenza
  - questo oggetto incapsula un insieme di parametri di configurazione – che sono definiti dall'amministratore del sistema
    - ad es., delivery mode (best effort o persistente) e supporto per transazioni, numero massimo di connessioni, timeout, numero massimo di tentativi per ottenere una connessione, rilevamento di messaggi duplicati, …
    - alcuni di questi parametri possono essere sovrascritti da proprietà dei messaggi individuali



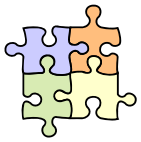
API di JMS 2.0 (1)

- Le nuove API di Java per il messaging (JMS 2.0), rispetto alle API precedenti (JMS 1.1), introducono alcune ulteriori interfacce, al fine di semplificare l'uso di JMS
  - in particolare, richiedono l'uso di un numero minore di oggetti con cui interagire
  - inoltre fanno uso di eccezioni "unchecked" anziché "checked"
- ***JMSContext***
  - combina in un singolo oggetto le funzionalità di una ***Connection*** e di una ***Session*** – poiché è comune usare una singola sessione per connessione
  - dunque, rappresenta una connessione virtuale tra un client e un provider JMS *e anche* un contesto single-threaded per la produzione e il consumo di messaggi – è anche una factory di messaggi, e di produttori e consumatori di messaggi



API di JMS 2.0 (2)

- ***JMSProducer***
  - un oggetto intermediario, di supporto, per inviare messaggi
  - a differenza di un ***MessageProducer***, può essere usato per inviare messaggi a più destinazioni
- ***JMSConsumer***
  - un oggetto intermediario, di supporto, per ricevere messaggi da una certa destinazione
  - come ***MessageConsumer***, incapsula una destinazione da cui ricevere 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 un contesto JMS

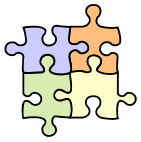
    context = connectionFactory.getContext(…);
  - usare il contesto come factory per creare un producer JMS

    messageProducer = context.createProducer();
  - usare il contesto come factory per creare un messaggio

    message = context.createTextMessage().setText(...);
  - usare il producer JMS per inviare il messaggio a una destinazione

    messageProducer.send(dest, message);
  - infine, chiudere il contesto (ovvero, la connessione)

    context.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 un contesto JMS

    context = connectionFactory.getContext(…);
  - usare il contesto come factory per creare un consumer JMS per una certa destinazione **dest**

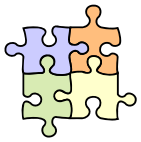
    messageConsumer = context.createConsumer(dest);
  - abilitare la ricezione di messaggi

    context.start();
  - usare il consumer per ricevere messaggi

    message = messageConsumer.receive();
  - arrestare la ricezione di messaggi e chiudere il contesto/la connessione

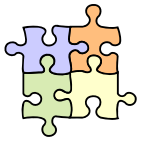
    context.stop();

    context.close();



Consumo di messaggi

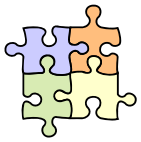
- Il consumo di messaggi può avvenire secondo due modalità
  - *consumo sincrono* – mostrato nell'esempio precedente – un consumatore legge i messaggi mediante un'operazione ***receive*** bloccante – dunque, qui "sincrono" va inteso come "bloccante"
  - *consumo asincrono* – in alternativa, un client consumatore può creare (e registrare) un oggetto consumatore che implementa un'interfaccia *message listener* – che deve implementare un metodo ***onMessage*** – che specifica 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 client consumatori interessati a ricevere messaggi da quella destinazione – e comunica il messaggio al consumatore selezionato invocando il metodo ***onMessage*** – in modo "asincrono" rispetto al consumatore



Consumo di messaggi

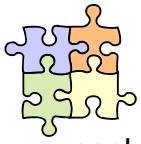
- Il consumo di messaggi può avvenire secondo due modalità
  - *consumo sincrono* – mostrato nell'esempio precedente – un consumatore legge i messaggi mediante un'operazione ***receive*** bloccante – dunque, qui "sincrono" va inteso come "bloccante"
  - *consumo asincrono* – in alternativa, un client consumatore può creare (e registrare) un oggetto consumatore che implementa un'interfaccia *message listener* – che deve implementare un metodo ***onMessage*** – che specifica cosa debba fare quando vi
    - il cli EE,
    - zione – e comunic selezionato invocando il metodo ***onMessage*** – in modo "asincrono" rispetto al consumatore

Attenzione: in ogni caso, la comunicazione tra produttore e consumatore è comunque asincrona



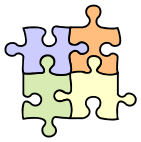
Esempi e configurazione

- Negli esempi che seguono si assume che
  - sia stato installato e configurato un provider JMS
    - ad esempio, Oracle GlassFish Server v4
  - 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



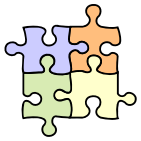
Esempio – produttore di messaggi (1)

```
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 (di solito remota) in una variabile
  - il compilatore riporta le annotazioni, metadati, nel bytecode – il compilatore non interpreta ulteriormente le annotazioni
  - le annotazioni sono normalmente prese in considerazione dagli strumenti di sviluppo (ad es., JUnit) e/o dall'ambiente di esecuzione – in questo caso, dall'ambiente/contenitore ("appclient") in cui viene eseguito l'application client
- In particolare, nell'esempio,
  - il valore di queue viene assegnato sulla base di una ricerca su un registry JNDI – prima che inizi l'esecuzione delle istruzioni dell'applicazione – chiaramente si tratta dell'iniezione di un proxy a una risorsa remota
  - attenzione, in un "application client" per Java EE, l'iniezione delle risorse avviene solo nella "main class"



Iniezione di risorse – in Java EE

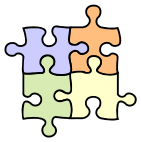
- Le annotazioni non vengono tradotte in istruzioni – ma in "annotazioni" del bytecode, poi interpretate dall'ambiente di sviluppo o di esecuzione
  - spesso semplificano il codice – in particolare, in questo caso rendono trasparente l'accesso al server JNDI per la ricerca delle varie risorse

Java EE 5-6-7

```
import javax.annotation.Resource;

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

Java EE 1.4

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

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

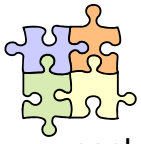



Esempio – produttore di messaggi (2)

- Anziché usare direttamente le API di JMS, usiamo una classe "connettore" **SimpleProducer** che incapsula l'accesso a JMS
  - attenzione, **SimpleProducer** non va confuso con **MessageProducer** o **JMSProducer**

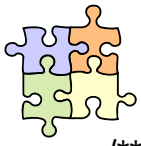
```
public static void main(String[] args) {
   /* crea il producer (per la coda) */
   SimpleProducer simpleProducer =
      new SimpleProducer("Produttore", queue, connectionFactory);

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

   /* invia alcuni messaggi */
   for (int i=0; i<10; i++) {
      ... produce un messaggio message ...
      simpleProducer.sendMessage(message);
   }

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

Qui va definita la logica applicativa del produttore.



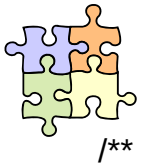
Esempio – produttore di messaggi (3)

```
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;

    /* contesto jms */
    private JMSContext context = null;
    /* per l'invio di messaggi alla destinazione */
    private JMSProducer messageProducer = null;

    ...
}
```

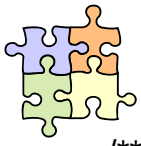
Esempio – produttore di messaggi (4)

```
/** 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;
}
```



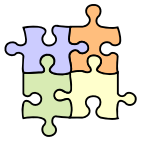
Esempio – produttore di messaggi (5)

```
/** Si connette alla destinazione JMS.
 * Crea anche un message producer. */
public void connect() {
    context = connectionFactory.createContext();
    messageProducer = context.createProducer();
}

/** Si disconnette dalla destinazione JMS. */
public void disconnect() {
    if (context != null) {
        context.close();
        context = null;
    }
}
```

Esempio – produttore di messaggi (6)

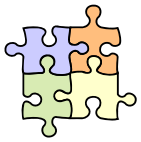
```
/** Invia un messaggio text alla destinazione. */
public void sendMessage(String text) {
    try {
        /* crea il messaggio */
        TextMessage message = context.createTextMessage();
        message.setText(text);
        /* invia il messaggio alla destinazione */
        messageProducer.send(destination, message);
     } catch (JMSException 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 {

    @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) {
        ...
    }
}
```

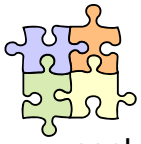
come prima



Esempio – consumatore sincrono (2)

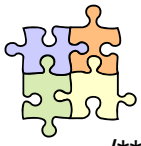
```
public static void main(String[] args) {

    /* 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 messaggi */
    while (true) {
        String message = simpleConsumer.receiveMessage();
        ... fa qualcosa con il messaggio message ricevuto ...
    }

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

Qui va definita la logica applicativa del consumatore.



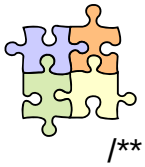
Esempio – consumatore sincrono (3)

```
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;

    /* contesto jms */
    private JMSContext context = null;
    /* per la ricezione dei messaggi dalla destinazione */
    private JMSConsumer messageConsumer = null;

    ...
}
```

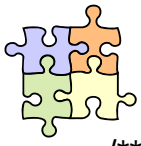
Esempio – consumatore sincrono (4)

```
/** 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;
}

...
```



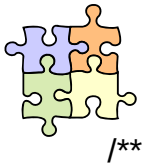
Esempio – consumatore sincrono (5)

```
/** Si connette alla destinazione JMS.
 * Crea anche un message consumer per la destinazione. */
public void connect() {
    context = connectionFactory.createContext();
    messageConsumer = context.createConsumer(destination);
}

... disconnect, come per il produttore ...

...
```

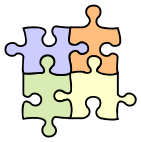
Esempio – consumatore sincrono (6)

```
/** Avvia la ricezione dei messaggi */
public void start() {
    /* avvia la consegna di messaggi per la connessione */
    context.start();
}

/** Arresta la ricezione dei messaggi */
public void stop() {
     /* arresta la consegna di messaggi per la connessione */
     context.stop();
}
```



Esempio – consumatore sincrono (7)

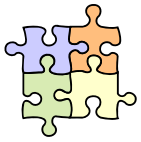
```
/** Riceve un messaggio dalla destinazione */
public String receiveMessage() {
    try {
        Message m = messageConsumer.receive();  // bloccante
        if (m instanceof TextMessage) {
            TextMessage message = (TextMessage) m;
            return message.getText();
        }
    } catch (JMSException e) {
        System.out.println("Error receiving message: " + e.toString());
        System.exit(1);
    }
    return null;
}
```

Esempio – consumatore asincrono (1)

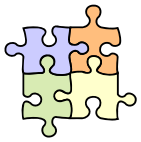
```
package asw.asw840.simpleasynchconsumer;

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

public class Main {

    @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) {
        ...
    }
}
```

come prima



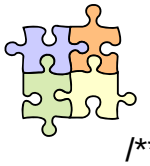
Esempio – consumatore asincrono (2)

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

    /* crea l'oggetto per elaborare i messaggi ricevuti */
    MessageListener simpleMessageProcessor =
        new SimpleMessageProcessor("Consumatore");

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

    /* 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)

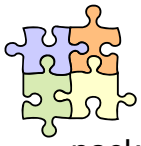
```
package asw.asw840.simpleasynchconsumer;

import javax.jms.*;

public class SimpleMessageProcessor implements MessageListener {

    /* nome di questo elaboratore di messaggi */
    String name;

    /** Crea un nuovo SimpleMessageProcessor di nome m. */
    public SimpleMessageProcessor(String n) {
        this.name = n;
    }

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

    /** Definisce la logica applicativa associata alla ricezione di un messaggio. */
    private void processMessage(String message) { ... }

}
```



Esempio – consumatore asincrono (4)

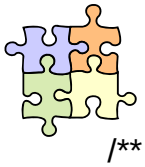
```
/** Riceve un messaggio. (Implementa MessageListener.) */
public void onMessage(Message m) {
    if (m instanceof TextMessage) {
        TextMessage textMessage = (TextMessage) m;
        try {
            this.processMessage( textMessage.getText() );
        } catch (JMSException e) {
         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

/** Definisce la logica applicativa associata alla ricezione di un messaggio. */
private void processMessage(String message) {
    ... fa qualcosa con il messaggio message ricevuto ...
}
```

Qui va definita la logica applicativa del consumatore.



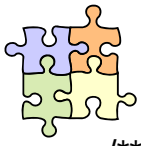
Esempio – consumatore asincrono (5)

```
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;
    /* listener associato a questo consumer */
    private MessageListener listener;

    /* contesto jms */
    private JMSContext context = null;
    /* per la ricezione dei messaggi dalla destinazione */
    private JMSConsumer messageConsumer = null;

    ...
}
```



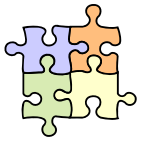
Esempio – consumatore asincrono (6)

```
/** Crea un nuovo SimpleAsynchConsumer, di nome n, per una Destination d,
 *   che delega l'elaborazione dei messaggi al listener l. */
public SimpleAsynchConsumer(String n, Destination d, ConnectionFactory cf,
                                        MessageListener l) {
    this.name = n;
    this.destination = d;
    this.connectionFactory = cf;
    this.listener = l;
}

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

...
```

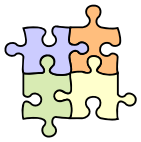
Esempio – consumatore asincrono (7)

```
/** Riceve messaggi dalla destinazione */
public void receiveMessages() {
    /* registra l'ascoltatore di messaggi */
    messageConsumer.setMessageListener(this.listener);
    /* avvia la consegna di messaggi */
    context.start();
    while (true) {
        /* mentre aspetta messaggi, non fa niente */
        /* probabilmente utile una variabile di controllo
         * e un "break" per smettere di accettare messaggi */
    }
    /* termina la consegna di messaggi */
    context.stop();
}
```



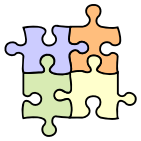
Il metodo onMessage()

- Chi invoca il metodo **onMessage**? Quando?
  - si consideri una destinazione (ad esempio, una 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 dal provider JMS, ad esempio da un application server (AS)
    - è l'AS che sa quali consumatori sono registrati (abbonati) presso una certa destinazione – ed è l'AS che decide a chi (C1 oppure C2) consegnare il messaggio M – e la consegna avviene invocando il metodo **onMessage** dell'oggetto listener associato a uno di questi due componenti



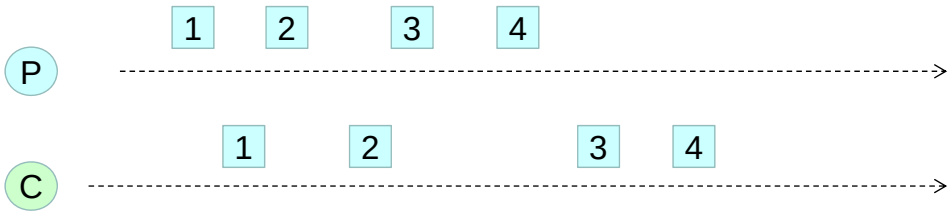
Osservazioni

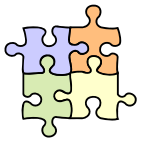
- Sicuramente è possibile fare numerose modifiche e miglioramenti al codice – ad esempio
  - miglior separazione tra componenti e connettori
  - maggior indipendenza dalla particolare tecnologia
    - ad es., un'interfaccia "listener" indipendente da JMS
  - classi di utilità per separare/mettere a fattor 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 meccanismo per consentire la terminazione dei consumatori
    - ad esempio, un messaggio che indica la "fine della sessione"
  - il caso più realistico prevede la presenza di più destinazioni JMS e di numerosi client JMS



- Alcuni esperimenti con le code

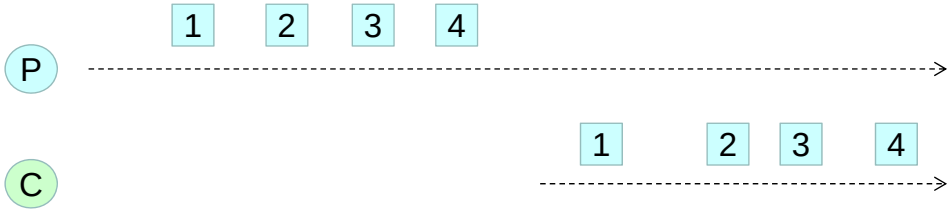
- Esperimento A con una coda
  - avvio il consumatore C, poi avvio il produttore P che invia N messaggi
  - conseguenze
    - il produttore invia N messaggi
    - il consumatore riceve N messaggi

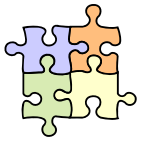




Alcuni esperimenti con le code

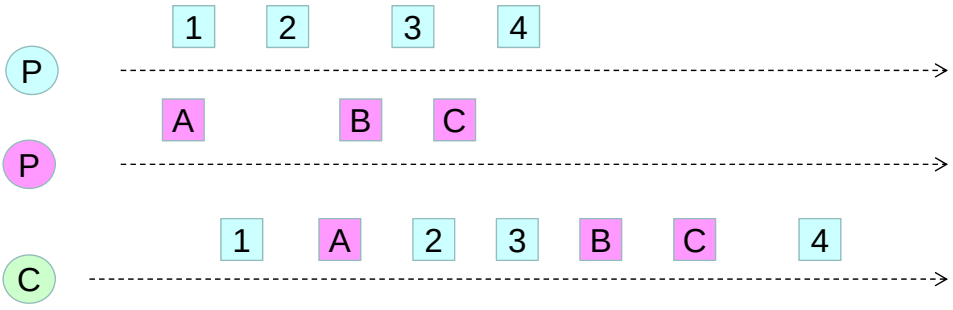
- Esperimento B con una coda
  - il consumatore non è inizialmente attivo, il produttore invia N messaggi
    - il produttore invia N messaggi, e termina
  - poi viene avviato il consumatore
  - conseguenze
    - il consumatore riceve N messaggi – a meno che sia passato un tempo tale da far “scadere”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

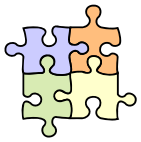




Alcuni esperimenti con le code

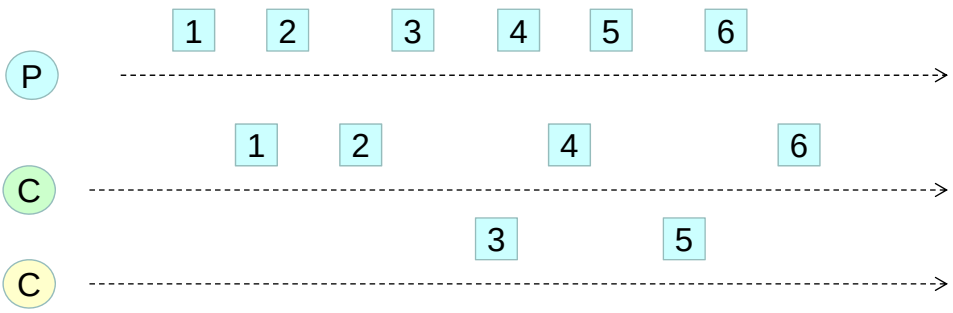
- Esperimento C con una coda
  - avvio il consumatore, e due produttori sulla stessa coda
  - conseguenze
    - un produttore invia N messaggi
    - un produttore invia M messaggi
    - il consumatore riceve N+M messaggi

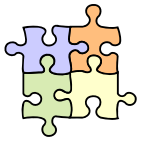




Alcuni esperimenti con le code

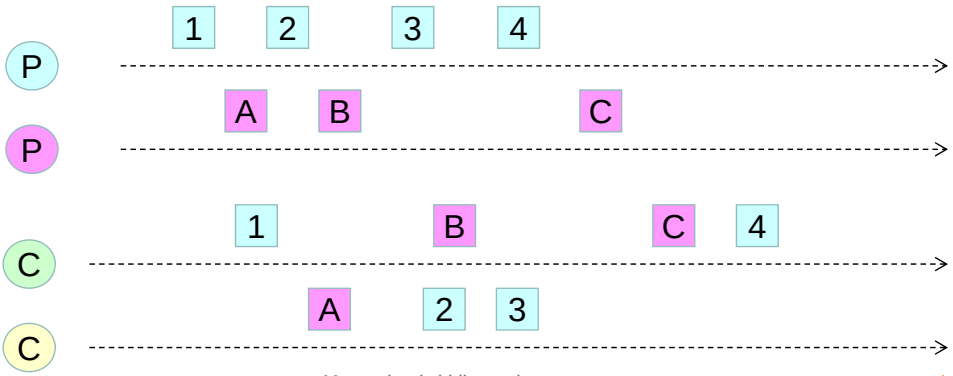
- Esperimento D con una coda
  - avvio due consumatori sulla stessa coda, poi il produttore invia N messaggi
  - conseguenze
    - il produttore invia N messaggi
    - un consumatore riceve X messaggi
    - l'altro consumatore riceve gli altri N-X messaggi

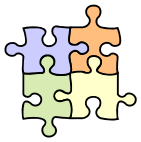




Alcuni esperimenti con le code

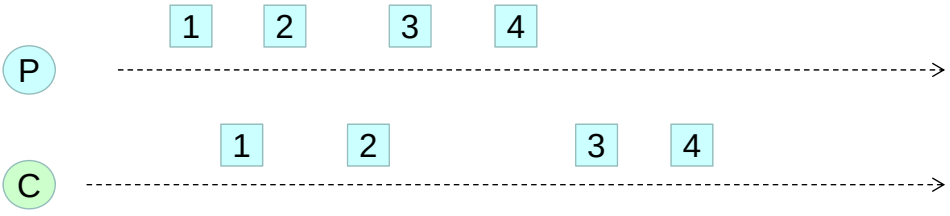
- Esperimento E con una coda
  - avvio due consumatori sulla stessa coda, poi due produttori sulla stessa coda
  - conseguenze
    - un produttore invia N messaggi, l'altro produttore ne invia M
    - un consumatore riceve X messaggi
    - l'altro consumatore riceve gli altri N+M-X messaggi

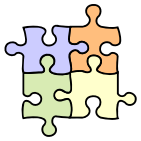




Alcuni esperimenti con gli argomenti

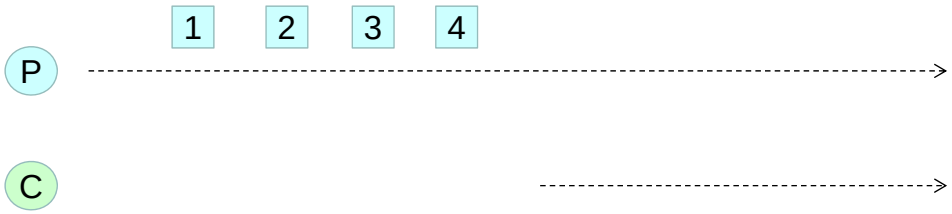
- Esperimento A con gli argomenti
  - avvio un consumatore, poi avvio il produttore che invia N messaggi
  - conseguenze
    - il produttore invia N messaggi
    - il consumatore riceve N messaggi

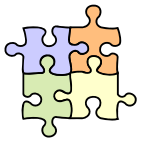




Alcuni esperimenti con gli argomenti

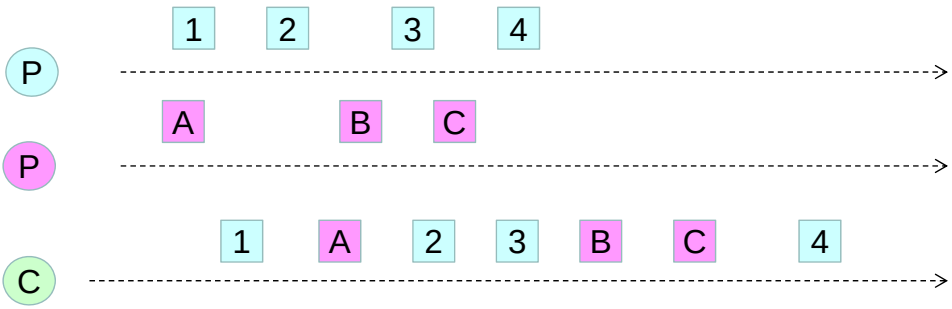
- Esperimento B con gli argomenti
  - il consumatore non è inizialmente attivo, il produttore invia N messaggi
    - il produttore invia N messaggi, e termina
  - poi viene avviato il consumatore
  - conseguenze
    - il consumatore non riceve nessun messaggio

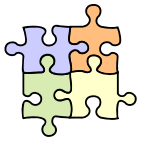




Alcuni esperimenti con gli argomenti

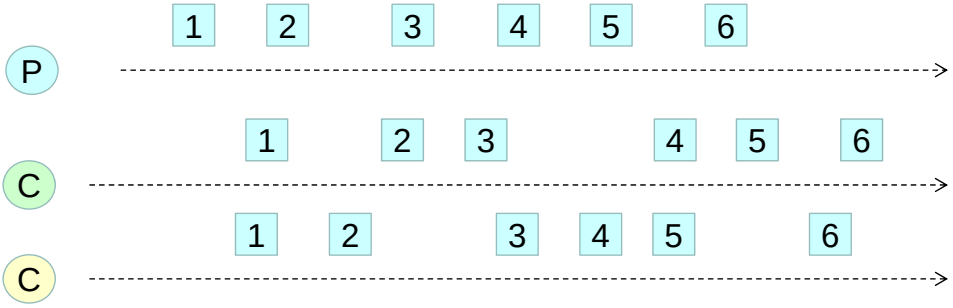
- Esperimento C con gli argomenti
  - avvio un consumatore, e due produttori sullo stesso argomento
  - conseguenze
    - un produttore invia N messaggi
    - un produttore invia M messaggi
    - il consumatore riceve N+M messaggi

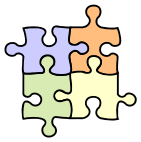




Alcuni esperimenti con gli argomenti

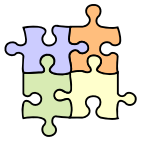
- Esperimento D con gli argomenti
  - avvio due consumatori sullo stesso argomento, poi un produttore invia N messaggi
  - conseguenze
    - il produttore invia N messaggi
    - ciascuno dei consumatori riceve N messaggi





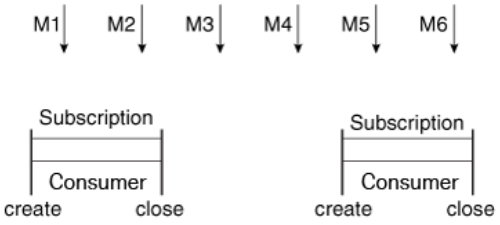
- Argomenti e registrazioni durature

- Nell'uso dei topic, sussiste una dipendenza temporale nell'invio/ricezione di messaggi
  - un client abbonato riceve normalmente solo i messaggi inviati al topic limitatamente al periodo di tempo in cui è registrato al topic ed attivo
- Questa dipendenza può essere rilassata tramite l'uso di "abbonamenti duraturi" (*durable subscription*) da parte dei client consumatori
  - in questo caso, a ciascuna subscription è associata un'identità univoca – un consumatore può ricevere messaggi per "sessioni", con riferimento all'identificatore della subscription
  - nelle sessioni successive, il consumatore potrà ricevere anche i messaggi che sono stati inviati nel periodo di tempo in cui è stato inattivo

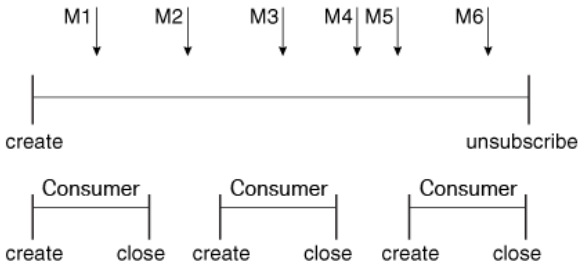


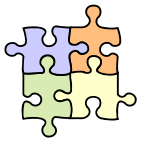
Registrazioni durature e non durature

- La registrazione (subscription) e il consumatore sono entità diverse – in particolare, la registrazione è un concetto lato server, mentre il consumatore è un client
  - registrazione non duratura – i messaggi M3 e M4 non sono ricevuti



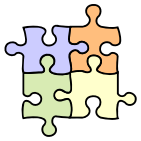
  - registrazione duratura – tutti i messaggi vengono ricevuti





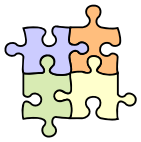
- Messaging e Pipes & Filters

- 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



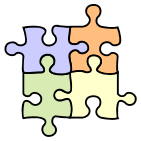
Messaging e Pipes & Filters

- Si vuole realizzare un'applicazione di tipo pipes and filters, in cui
  - ma come realizzare ciascun filtro? come fare in modo che si comporti sia da consumatore che da produttore di messaggi?
  - nel caso più semplice, un filtro deve produrre un messaggio M_{out} per ciascun messaggio consumato M_{in}
  - al filtro viene associato sia un consumatore di messaggi (ad esempio, un oggetto **SimpleConsumer**) che un produttore di messaggi (ad esempio, un **SimpleProducer**)
  - nel metodo **processMessage** del consumatore di messaggi, che elabora il messaggio M_{in} (o comunque, a partire dal metodo **onMessage**), bisogna poi specificare
    - come generare il messaggio M_{out} da M_{in}
    - l'invio del messaggio M_{out} alla destinazione successiva – tramite il produttore di messaggi



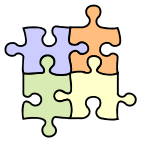
- Struttura dei messaggi

- In generale, ogni messaggio può avere tre parti
  - un'intestazione (*header*) – l'unica parte obbligatoria, che contiene un insieme di campi predefiniti
  - un insieme di *proprietà* – aggiuntive e personalizzate, rispetto a quelle dell'intestazione
  - un corpo (*body*) – il messaggio può essere di diversi tipi, ad esempio
    - un messaggio di testo
    - un messaggio binario
    - una mappa, ovvero un insieme di coppie nome-valore
    - un oggetto serializzato
    - un messaggio vuoto – tutte le informazioni sono nell'intestazione



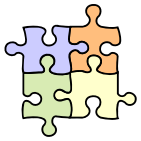
Struttura dei messaggi

- L'header di un messaggio contiene un insieme di campi predefiniti
  - un message ID
  - un correlation ID
    - ad es., un messaggio di risposta indicherà 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, creata a runtime
  - la priorità
  - una modalità di consegna – persistente o non persistente
  - un tempo di expiration
  - un ritardo nella consegna del messaggio



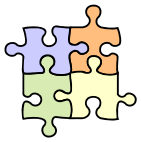
Esempio – scambio di messaggi

- Ad esempio, per realizzare uno scambio di messaggi richiesta/risposta
  - potrebbe essere usata una coda per le richieste e una coda separata per le risposte
- Ma come gestire il caso in cui N produttori (client) inviano richieste ad M consumatori (server) – ma poi vogliono delle risposte indirizzate espressamente a loro?
  - ciascun produttore (client) potrebbe creare (a runtime) una propria destinazione temporanea
  - il produttore (client) potrebbe poi specificare nel proprio messaggio di richiesta il riferimento a questa destinazione temporanea nel campo ReplyTo – in modo che il consumatore (server) possa inviare la risposta alla richiesta proprio a quella destinazione temporanea



- Affidabilità

- La modalità di consegna dei messaggi può essere
  - non persistente – può migliorare prestazioni e scalabilità, ma è possibile la perdita di messaggi se il provider JMS fallisce
  - persistente (il default)
- Acknowledgement
  - un messaggio viene considerato consumato con successo solo al momento dell'ack – che avviene dopo la ricezione e l'elaborazione del messaggio – l'ack può essere iniziato dal client oppure automatico (il default)
- Inoltre, un client può usare una transazione per ricevere e/o inviare un messaggio o un gruppo di messaggi in modo atomico
  - notifica del consumo effettivo dei messaggi (ack) e invio effettivo dei messaggi solo al momento del commit
  - questa transazioni possono essere anche combinate con transazioni più ampie, ad esempio di basi di dati



- 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