Messaging
Parte 1

Programmazione Distribuita - A.A. 2020/2021

Biagio Cosenza
Dipartimento di Informatica
Università di Salerno
http://cosenza.eu/
bcosenza@unisa.it

Organizzazione della Lezione

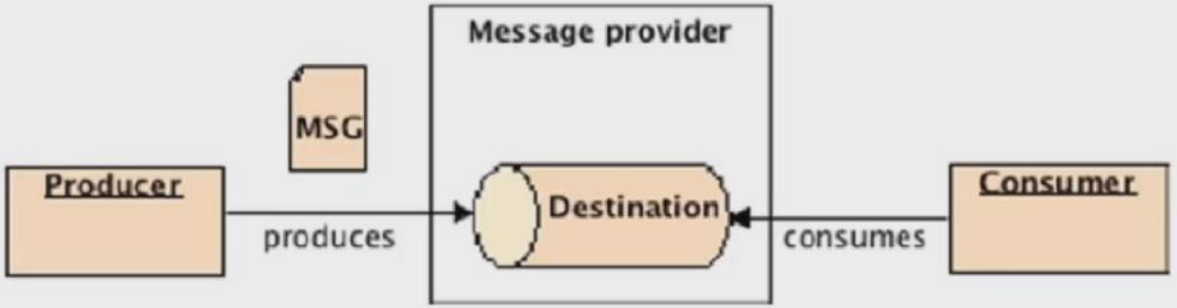
- Introduzione
- Messaging
- Java Messaging Service API
- Message Producers
- Message Consumers
- Conclusioni

Introduzione: Message-Oriented Middleware

- MOM (Message-oriented middleware) è un software (provider) che permette lo scambio di messaggi asincroni fra sistemi eterogenei
- Può essere visto come un buffer che produce e consuma messaggi
- È intrinsecamente *loosely coupled*
  - che i produttori non sanno chi è all'altra estremità del canale di comunicazione a consumare il messaggio
  - Il produttore e il consumatore non devono essere disponibili contemporaneamente per comunicare

Introduzione: Provider, Producer, Consumer

- Quando un messaggio viene inviato, il software che memorizza il messaggio e lo invia è detto Provider (a volte chiamato *broker*)
- Il sender del messaggio è chiamato Producer e la locazione in cui il messaggio è memorizzato è detta destinazione
- La componente che riceve il messaggio è detta Consumer
- Ogni componente interessata ad un messaggio in una particolare destinazione può consumarlo

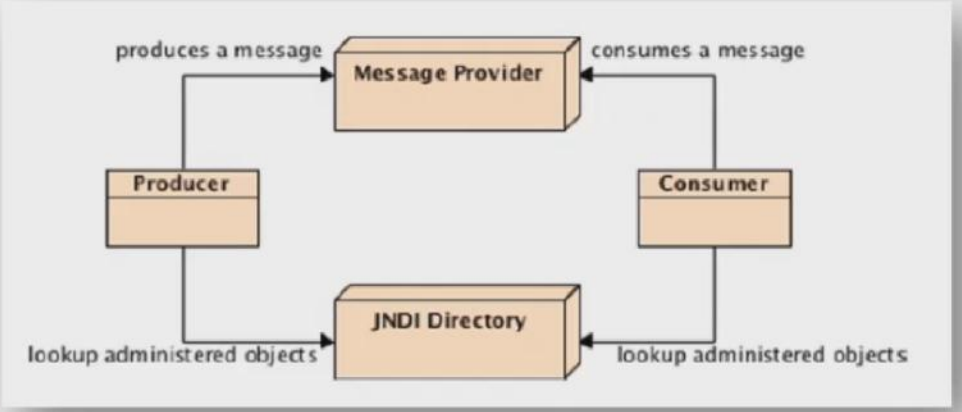


Java Message Service (JMS)

- In Java EE, l'API che gestisce questi concetti è Java Message Service (JMS)
- Set di interfacce e classi per
  - connettersi ad un provider
  - creare un messaggio
  - inviare un messaggio
  - ricevere un messaggio
- In un EJB container, *Message-Driven Beans* (MDBs) possono essere usati per ricevere messaggi in maniera *container-managed*

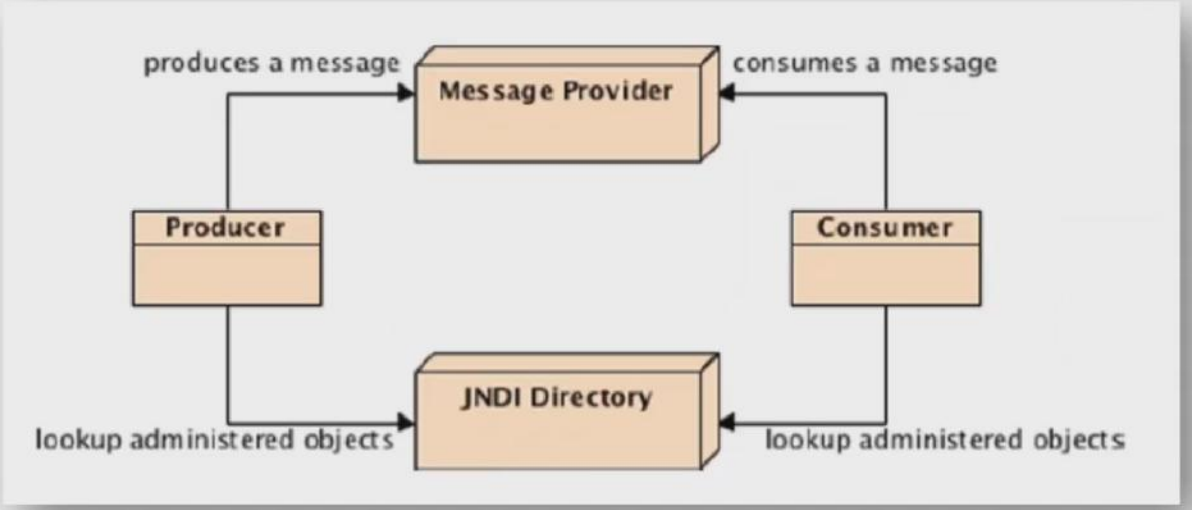
Architettura di Messaging

- Componenti di un'architettura di messaging:
  - un Provider: componente necessaria per instradare messaggi
    - gestisce buffering e delivery dei messaggi
  - i Clients: una qualunque applicazione Java o una componente che produce o consuma messaggi per/da un provider
    - il termine *Client* si usa genericamente per producer, sender, publisher, consumer, receiver, subscriber



Architettura di Messaging

- Componenti di un'architettura di messaging:
  - Messages: oggetti che i client inviano/ricevono dal provider
  - Administered objects: oggetti (connection factories e destinazioni) fornite attraverso JNDI lookups o injection

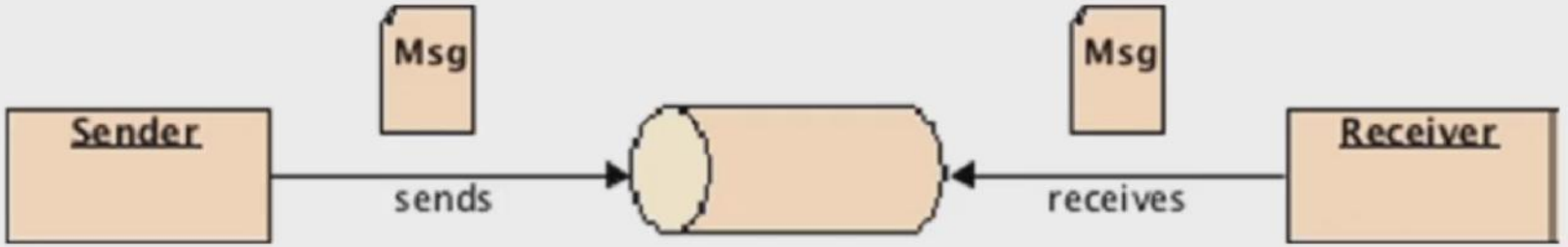


Architettura di Messaging

- Il Provider permette comunicazione asincrona fornendo una destinazione dove i messaggi possono essere mantenuti finché non vengono instradati verso un client
- Esistono due differenti tipi di destination:
  - Point-to-point (P2P) model: la destination è chiamata coda
    - il client inserisce un messaggio in coda, mentre un altro client riceve il messaggio
    - una volta fatto acknowledge, il message provider rimuove il messaggio dalla coda
  - Publish-subscribe (pub-sub) model: la destination è chiamata topic
    - il client pubblica un messaggio con un topic, e tutti i sottoscrittori al topic riceveranno il messaggio

Modello Point-to-Point

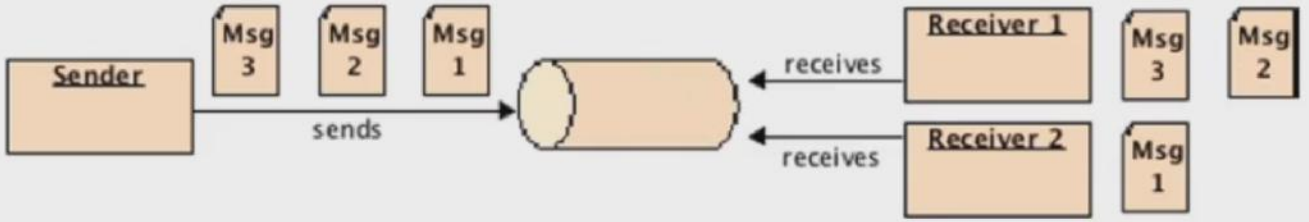
- Il messaggio viaggia da un singolo producer verso un singolo consumer



- Ogni messaggio viene inviato ad una specifica coda, ed il receiver riceve il messaggio dalla coda
- La coda mantiene i messaggi finché non vengono consumati o scadono

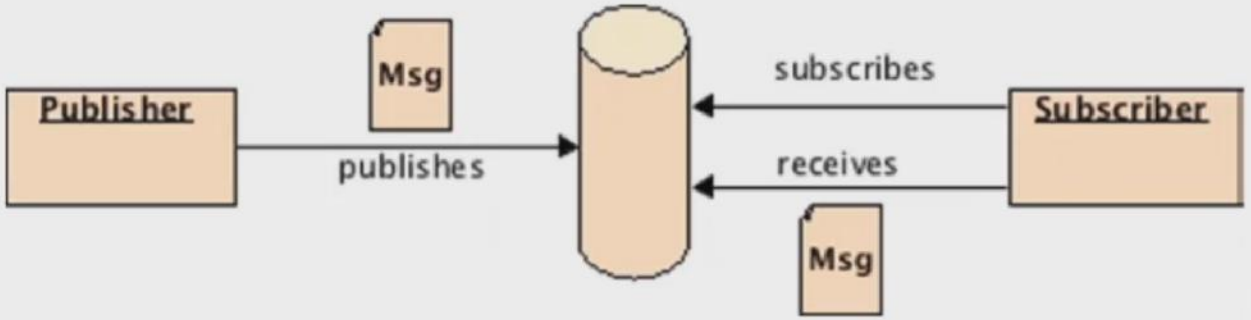
Modello Point-to-Point

- Nel modello P2P, esiste un solo receiver per ogni messaggio
- Una coda può avere consumers multipli
  - ma quando un receiver consuma il messaggio, questo viene tolto dalla coda, e nessun altro receiver potrà consumarlo
- Il modello P2P non garantisce che i messaggi siano instradati in un particolare ordine
  - un provider può riceverli in un particolare ordine, o random, o in qualunque altro ordine



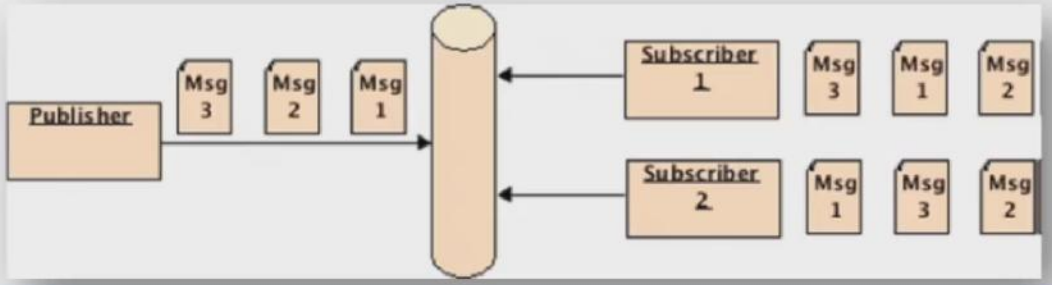
Modello Publish-Subscriber

- Mel modello pub-sub model, un singolo messaggio è inviato da un singolo producer per potenzialmente diversi consumers
- Il modello è costruito intorno ai concetti di topics, publishers, e subscribers
  - I Consumers sono chiamati subscribers
    - Hanno necessità di sottoscriversi ad un topic
    - Forniscono il meccanismo di subscribing/unsubscribing, che occorre dinamicamente



Modello Publish-Subscriber

- Il topic conserva i messaggi fino a quando non vengono distribuiti a tutti i subscribers
- Dipendenza temporale fra publisher e subscriber
  - i subscribers NON ricevono i messaggi inviati PRIMA della loro sottoscrizione
  - se il subscriber è inattivo per un periodo di tempo determinato, esso non riceve messaggi vecchi quando diventa nuovamente attivo
- Multipli subscribers possono consumare lo stesso messaggio
- Utile per applicazioni di tipo *broadcast*:
  - singolo messaggio recapitato a diversi consumatori



Adminstered Objects

- Oggetti che si configurano amministrativamente, e non programmatically
- Il provider permette di configurare questi oggetti e li rende disponibili nello spazio dei nomi JNDI
- Come i JDBC datasources, questi oggetti vengono creati solo una volta
- I due tipi di oggetti amministrati sono:
  - connection factory: usato dai clienti per creare una connessione a una destinazione
  - destinazioni: punti di distribuzione del messaggio che ricevono, mantengono, e distribuiscono messaggi
    - le destinazioni possono essere code (P2P) o topic (pub-sub)

Message-Driven Beans

- Message-Driven Beans (MDBs) sono message consumer asincroni eseguiti in un EJB container
  - l'EJB container si occupa dei servizi (transactions, security, concurrency, message acknowledgment, etc.), mentre l'MDB si occupa di consumare messaggi
- I MDB sono stateless
  - l'EJB container può avere numerose istanze, eseguite in concorrenza per processare messaggi provenienti da diversi producers
  - non mantengono stato attraverso invocazioni separate
- In generale gli MDBs sono in ascolto su una destination (queue o topic) e, quando il messaggio arriva, lo consuma e lo processa

Message-Driven Beans

- I MDBs rispondono a messaggi ricevuti dal container
  - laddove gli stateless session beans rispondono a richieste client attraverso una interfaccia appropriata (local, remote, o no-interface)

Java Messaging Service API

- JMS è un insieme di standard Java API che permette alle applicazioni di creare, inviare, ricevere e leggere messaggi in maniera asincrona
- Definisce un insieme di interfacce e classi per la comunicazione con altri message providers
- JMS è analogo a JDBC:
  - JDBC permette la connessione a differenti databases (Derby, MySQL, Oracle, DB2, etc.)
  - JMS permette la connessione a diversi providers (OpenMQ, MQSeries, SonicMQ, etc.)

Connection Factory

- Connection factories sono administered objects
  - l'interfaccia `javax.jms.ConnectionFactory` incapsula i parametri definiti da un amministratore
  - per usare un administered object come una ConnectionFactory, il client deve eseguire una JNDI lookup (o usare injection)
- Esempio: si ottiene un JNDI InitialContext object e lo si usa per fare lookup di una connectionFactory attraverso il suo nome JNDI:

```
Context ctx = new InitialContext();
ConnectionFactory ConnectionFactory = ➥
                (ConnectionFactory) ctx.lookup("jms/javaee7/ConnectionFactory");
```

Destination

- Una destination è un administered object che contiene provider-specific configuration information
  - come ad esempio un destination address
- Questo meccanismo è nascosto al JMS client attraverso l'uso dell'interfaccia `javax.jms.Destination`
- Come per le connection factory, una JNDI lookup è necessaria per restituire tali oggetti:

```
Context ctx = new InitialContext();
Destination queue = (Destination) ctx.lookup("jms/javaee7/Queue");
```

Connection Factory

- Connection factories sono administered objects
  - l'interfaccia `javax.jms.ConnectionFactory` incapsula i parametri definiti da un amministratore
  - per usare un administered object come una ConnectionFactory, il client deve eseguire una JNDI lookup (o usare injection)
- Esempio: si ottiene un JNDI InitialContext object e lo si usa per fare lookup di una connectionFactory attraverso il suo nome JNDI:

```
Context ctx = new InitialContext();
ConnectionFactory ConnectionFactory = ➥
                (ConnectionFactory) ctx.lookup("jms/javaee7/ConnectionFactory");
```

Destination

- Una destination è un administered object che contiene provider-specific configuration information
  - come ad esempio un destination address
- Questo meccanismo è nascosto al JMS client attraverso l'uso dell'interfaccia `javax.jms.Destination`
- Come per le connection factory, una JNDI lookup è necessaria per restituire tali oggetti:

```
Context ctx = new InitialContext();
Destination queue = (Destination) ctx.lookup("jms/javaee7/Queue");
```

Simplified API

- Le altre interfacce:
  - `JMSContext`
    - una connessione attiva ad un JMS provider e un context single-threaded per inviare e ricevere messaggi
  - `JMSProducer`
    - oggetto creato da un JMSContext per inviare messaggi ad una coda o ad un topic
  - `JMSConsumer`
    - oggetto creato da un JMSContext per ricevere messaggi inviati ad una coda o ad un topic

JMS Context API

Property	Description
`void start()`	Starts (or restarts) delivery of incoming messages
`void stop()`	Temporarily stops the delivery of incoming messages
`void close()`	Closes the `JMSContext`
`void commit()`	Commits all messages done in this transaction and releases any locks currently held
`void rollback()`	Rolls back any messages done in this transaction and releases any locks currently held
`BytesMessage createBytesMessage()`	Creates a `BytesMessage` object
`MapMessage createMapMessage()`	Creates a `MapMessage` object
`Message createMessage()`	Creates a `Message` object
`ObjectMessage createObjectMessage()`	Creates an `ObjectMessage` object
`StreamMessage createStreamMessage()`	Creates a `StreamMessage` object
`TextMessage createTextMessage()`	Creates a `TextMessage` object
`Topic createTopic(String topicName)`	Creates a `Topic` object
`Queue createQueue(String queueName)`	Creates a `Queue` object
`JMSConsumer createConsumer(Destination destination)`	Creates a `JMSConsumer` for the specified destination
`JMSConsumer createConsumer(Destination destination, String messageSelector)`	Creates a `JMSConsumer` for the specified destination, using a message selector
`JMSProducer createProducer()`	Creates a new `JMSProducer` object which can be used to configure and send messages
`JMSContext createContext(int sessionMode)`	Creates a new `JMSContext` with the specified session mode

JMS Producer API

Property	Description
`get/set[Type]Property`	Sets and returns a message property where `[Type]` is the type of the property and can be `Boolean`, `Byte`, `Double`, `Float`, `Int`, `Long`, `Object`, `Short`, `String`
`JMSProducer clearProperties()`	Clears any message properties set
`Set<String> getPropertyNames()`	Returns an unmodifiable `Set` view of the names of all the message properties that have been set
`boolean propertyExists(String name)`	Indicates whether a message property with the specified name has been set
`get/set[Message Header]`	Sets and returns a message header where `[Message Header]` can be `DeliveryDelay`, `DeliveryMode`, `JMSCorrelationID`, `JMSReplyTo`, `JMSType`, `Priority`, `TimeToLive`
`JMSProducer send(Destination destination, Message message)`	Sends a message to the specified destination, using any send options, message properties and message headers that have been defined
`JMSProducer send(Destination destination, String body)`	Sends a `TextMessage` with the specified body to the specified destination

JMS Consumer API

Property	Description
`void close()`	Closes the `JMSConsumer`
`Message receive()`	Receives the next message produced
`Message receive(long timeout)`	Receives the next message that arrives within the specified timeout interval
`<T> T receiveBody(Class<T> c)`	Receives the next message produced and returns its body as an object of the specified type
`Message receiveNoWait()`	Receives the next message if one is immediately available
`void setMessageListener(MessageListener listener)`	Sets the `MessageListener`
`MessageListener getMessageListener()`	Gets the `MessageListener`
`String getMessageSelector()`	Gets the message selector expression

Message Producers

- Con le nuove API di JMS 2.0, la scrittura di produttori e consumatori diventa meno verbosa
- Rimangono gli administered objects `ConnectionFactory` e `Destination`
- A seconda se si è fuori o dentro un container, si usa JNDI lookup oppure injection
- Tre esempi:
  1. Produttore fuori da un container
  2. Produttore in un container
  3. Produttore in un container con CDI

Message Producers

- Con le nuove API di JMS 2.0, la scrittura di produttori e consumatori diventa meno verbosa
- Rimangono gli administered objects `ConnectionFactory` e `Destination`
- A seconda se si è fuori o dentro un container, si usa JNDI lookup oppure injection
- Tre esempi:
  1. Produttore fuori da un container
  2. Produttore in un container
  3. Produttore in un container con CDI

Message Producers (1): Produttore fuori da un Container

- Un oggetto `JMSProducer` viene creato da un `JMSContext` e usato per inviare messaggi

- I passi da seguire:
  1. Ottenere una connection factory ed una coda con JNDI
  2. Creare un `JMSContext` usando la factory
  3. Creare un `JMSProducer` usando il contesto
  4. Inviare un messaggio usando il metodo `send()` del producer

Message Producers (1): Produttore fuori da un Container

- Esempio: una classe Producer produce un Message in una Queue
  - prende il contesto da JNDI / cerca per l'administered object di `ConnectionFactory`
  - preleva la coda dal JNDI
  - cerca di creare il contesto; se fallisce chiude il contesto
  - crea il produttore dal contesto e invia un messaggio

```
public class Producer {
  public static void main(String[] args) {
   try{
    Context jndiContext = new InitialContext();
    ConnectionFactory connectionFactory = (ConnectionFactory)
      jndiContext.lookup("jms/javaee7/ConnectionFactory");
    Destination queue = (Destination) jndiContext.lookup("jms/javaee7/Queue");
    try(JMSContext context = connectionFactory.createContext()) {
      context.createProducer().send(queue,"Text message sent at"+ new Date());
    }
   } catch(NamingException e) {
    e.printStackTrace();
   }
  }
}
```

Message Producers (2): Produttore in un Container

- Produttore in un Container: il produttore diventa un EJB
- Esempio: un ProducerEJB in esecuzione all'interno di un Container usando `@Resource`
  - EJB stateless / Annotazione di una risorsa da ricercare su JNDI per la connection factory e la coda
  - Rifattorizzazione del comportamento in un metodo di business

```
@Stateless
public class ProducerEJB {

  @Resource(lookup ="jms/javaee7/ConnectionFactory")
  private ConnectionFactory connectionFactory;

  @Resource(lookup ="jms/javaee7/Queue")
  private Queue queue;

  public void sendMessage() {
    try(JMSContext context = connectionFactory.createContext()) {
     context.createProducer().send(queue, "Text message sent at"+ new Date());
    }
  }
}
```

Message Producers (1): Produttore fuori da un Container

- Esempio: una classe Producer produce un Message in una Queue
  - prende il contesto da JNDI / cerca per l'administered object di `ConnectionFactory`
  - preleva la coda dal JNDI
  - cerca di creare il contesto; se fallisce chiude il contesto
  - crea il produttore dal contesto e invia un messaggio

```
public class Producer {
  public static void main(String[] args) {
   try{
    Context jndiContext = new InitialContext();
    ConnectionFactory connectionFactory = (ConnectionFactory)
      jndiContext.lookup("jms/javaee7/ConnectionFactory");
    Destination queue = (Destination) jndiContext.lookup("jms/javaee7/Queue");
    try(JMSContext context = connectionFactory.createContext()) {
      context.createProducer().send(queue,"Text message sent at"+ new Date());
    }
   } catch(NamingException e) {
    e.printStackTrace();
   }
  }
}
```

Message Producers (3): Produttore in un Container con CDI

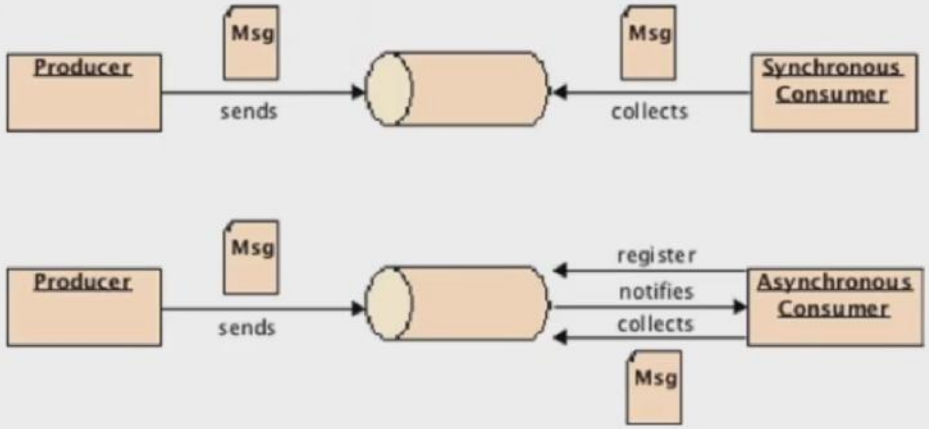
- Esempio di Managed Bean che produce un Message usando `@Inject`
  - Il container fa inject della factory di connessioni JMS, specificando il tipo
  - coda ricercata su JNDI / metodo di business

```
public class Producer {

  @Inject
  @JMSConnectionFactory("jms/javaee7/ConnectionFactory")
  private JMSContext context;

  @Resource(lookup ="jms/javaee7/Queue")
  private Queue queue;

  public void sendMessage() {
    context.createProducer().send(queue, "Text message sent at " + new Date());
  }
}
```

Message Consumers

- Tipi di Consumer

1. Sincroni: il ricevitore esplicitamente preleva il messaggio dalla destinazione, invocando `receive()`
2. Asincroni: il ricevitore di registra all'evento di arrivo di un messaggio e implementa `MessageListener`
   - che all'arrivo del messaggio viene invocato il suo metodo `onMessage()`



Message Consumers: Consumer Sincrono

- Esempio di classe Consumer che consuma Messages in modo sincrono
  - preleva il contesto JNDI / cerca gli oggetti amministrati (factory per le connessioni e coda)
  - acquisisce il contesto (con ciclo di vita gestito da try-with-resources) / prova a ricevere

```
public class Consumer {
  public static void main(String[] args) {
   try{
    Context jndiContext = new InitialContext();
    ConnectionFactory connectionFactory = (ConnectionFactory)
      jndiContext.lookup("jms/javaee7/ConnectionFactory");
    Destination queue = (Destination)
      jndiContext.lookup("jms/javaee7/Queue");

    try(JMSContext context = connectionFactory.createContext()) {
      while(true) {
        String message = context.createConsumer(queue).receiveBody(String.class);
      }
    }
   } catch(NamingException e) { e.printStackTrace(); }
  }
}
```

Message Consumers: Consumer Asincrono

- Esempio di classe Consumer che e' un Message Listener
  - implementa interfaccia per listener / preleva il contesto JNDI / lookup per gli oggetti administered
  - factory di connessioni JMS / crea un oggetto di tipo `Listener` e lo registra
  - all'arrivo di un messaggio, invoca la callback

```
public class Listener implements MessageListener {
  public static void main(String[] args) {
    try{
      Context jndiContext = new InitialContext();
      ConnectionFactory connectionFactory = (ConnectionFactory)
        jndiContext.lookup("jms/javaee7/ConnectionFactory");
      Destination queue = (Destination) jndiContext.lookup("jms/javaee7/Queue");
      try(JMSContext context = connectionFactory.createContext()) {
        context.createConsumer(queue).setMessageListener(new Listener());
      }
    } catch(NamingException e) { e.printStackTrace(); }
  }

  public void onMessage(Message message) {
    System.out.println("Async Message received:"+ message.getBody(String.class));
  }
}
```

Conclusioni

- Introduzione
- Messaging
- Java Messaging Service API
- Message Producers
- Message Consumers
- Conclusioni