Politecnico
di Milano

Java Enterprise Edition 7 (JEE 7)
Overview

Michele Guerriero
michele.guerriero@polimi.it

JEE & Testing Sessions

- Sessions 1 & 2: Java EE introduction and main services
- Session 3: software testing
- **Session 4: Java EE advanced topics**
  - Introduce new available services (JNDI, JMS) and their APIs
  - More in depth look at some of the JEE platform's internal mechanisms
- Java EE Lab: apply what you have seen in sessions 1 & 2

Java Naming and Directory Interface (JNDI) service

- JNDI is a naming service enabling components to locate other components or resources
- A **resource** can be a piece of software providing connections to systems (e.g. a JDBC resource) or any Java class on the classpath
- Each resource is identified by a unique, people-friendly name, called the JNDI name
  - Example: the name of the Glassfish default JDBC resource for the Derby Java DB database is *java:comp/DefaultDataSource*

JNDI APIs

- Applications can look up for resources by calling the JNDI service through the JNDI APIs

- Example:
  - ```
    DataSource ds= (DataSource)
    InitialContext.lookup("java:comp/DefaultData
    Source");
    ```

From *the javax.naming* package (JNDI API)
    ```

JEE Injection Mechanisms

- JEE containers can **inject** objects into applications at runtime
- In the application code the injected objects doesn't need to be explicitly created (no `new` keyword)
- Injectamion allows to **decouple** your code from the implementations of its dependencies
- Injection works by mean of annotations
- Two types of injection:
  - Resource Injection
  - Dependency Injection

Resource Injection

- **Resource injection** enables you to inject any resource available in the JNDI namespace into any container-managed object, such as a servlet or an EJB
- For example, you can use resource injection to inject data sources available in the JNDI namespace

```
public class MyServlet extends HttpServlet {
    @Resource(name="java:comp/DefaultDataSource")
    private javax.sql.DataSource dsc;
    ...
}
```

Instruct the servlet container to look up for the "java:comp/DefaultDataSource" named resource from the JNDI service and to provide the MyServlet object with an instance of the resolved resource

Dependency Injection

- **Dependency injection** allows to turn Java classes into **managed objects** (Java objects that are created and managed by containers) and to inject them into any other objects
- Dependency injection in Java EE defines **scopes**, which **determine the lifecycle** of the objects that the container creates and injects
- To define managed objects (also called **managed beans**) assign a **scope** to a regular class
- Then use the ***@Inject*** annotation to inject the defined managed beans
- As opposed to resource injection, dependency injection is **type safe** because it resolves by type

Dependency Injection Example

```
@javax.enterprise.context.RequestScoped
public class CurrencyConverter { ... }


public class MyServlet extends HttpServlet {
    @Inject CurrencyConverter cc;
    ...
}
```

- **@RequestScoped** declare the CurrencyConverter class as a managed bean whose scope is limited to a given request
- When a request is assigned to an object **A** of type MyServlet, the (web) container creates an object **B** of type CurrencyConverter and makes it available to (**inject** it into) **A**
- After the request has been processed, the container destroys **B**, being it “RequestScoped”

Managed Beans and EJBs

- An EJB (seen in lecture 2) is just a particular kind of the more general concept of managed bean
- They have specific lifecycles (remember the instance pooling and the activation/passivation mechanisms)
- They are managed by a specific EJB container
- They provide system-level services, such as transactions and security
- Can be injected with the @EJB annotation, which is quite similar to the @Inject annotation apart from few advanced features specific to EJBs
- They are supposed to be used in large, distributed applications (i.e. Enterprise Applications)

Main Differences Between Resource Injection and Dependency Injection

Table 4–1 Differences Between Resource Injection and Dependency Injection

Injection Mechanism	Can Inject JNDI Resources Directly	Can Inject Regular Classes Directly	Resolves By	Type Safe
Resource Injection	Yes	No	Resource name	No
Dependency Injection	No	Yes	Type	Yes

[From the JEE 7 tutorial]

Dependency injection is preferred when the component to inject is managed by the same container of the component that requires the injection.
Resource injection is the only alternative when this is not the case (e.g. a servlet in a Web Server which needs a reference to a managed bean, managed by an Application Server)

Messaging

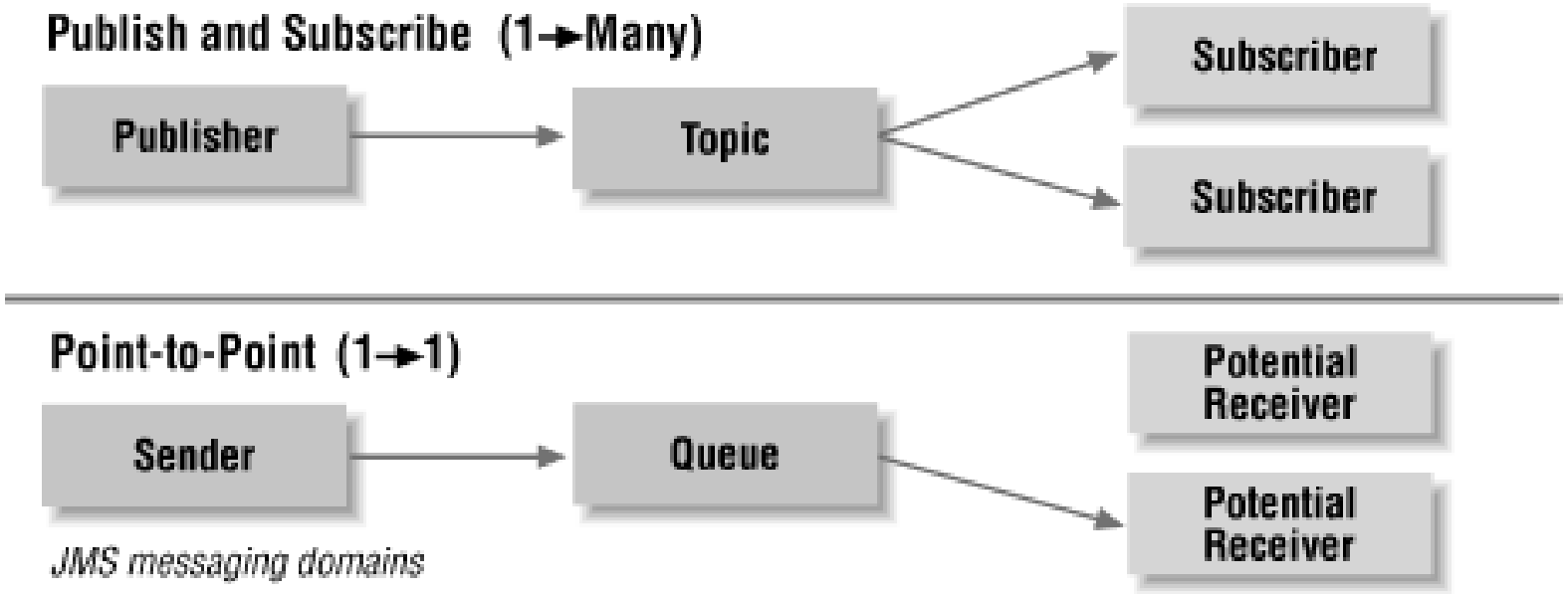
- Messaging is a method of communication between software components or applications
- A messaging client can send messages to, and receive messages from, any other client
- Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages
- Messaging enables distributed communication that is **loosely coupled**. The message destination is all that the sender and receiver have in common

Java Message Service (JMS)

- Java API specification that enables and simplify messaging in Java applications
- A JMS implementation acts as the agent (a.k.a. broker) of a messaging system
- Portability of JMS applications across JMS providers
- JMS enables communication that is not only loosely coupled but also **asynchronous** and **reliable**

JMS Messaging Styles

- JMS supports both point-to-point (**PTP**) and **publish-subscribe** messaging styles



JMS messaging domains

JMS API Programming Model – Aministered objects

- Aministered objects: **connection factories** and **destinations** usually injected using JNDI
- Managed administratively
- **Connection factory**: encapsulates a set of connection configuration properties that has been defined by an administrator:
  - `@Resource(lookup = "java:comp/DefaultJMSConnectionFactory") private static ConnectionFactory connectionFactory;`
- **Destination**: the object a client uses to specify the target of messages it produces and the source of messages it consumes
  - **Queue** for PTP, **Topic** for pub/sub;

JMS API Programming Model – the JMSContext

- **Connections**: created using a ConnectionFactory, it encapsulates a virtual connection with a JMS provider
- **Sessions**: created using a Connection, it is single-threaded context for producing and consuming messages
- **JMSContext**:
  - combines a Connection and a Session in a single object
  - created using a ConnectionFactory:

```
JMSContext context =
connectionFactory.createContext();
```

JMS API Programming Model – Producers

- Created using a JMSContext or a session and used for sending messages to a destination

```
JMSProducer producer = context.createProducer();
producer.send(dest, message);
```

JMS API Programming Model – Consumers

- Created using a JMSContext or a session and used for receiving messages from a destination

```
JMSConsumer consumer = context.createConsumer(dest);
Message m = consumer.receive();
```

- By default message delivery begins as soon as you have created the consumer (assuming the connection is started)
- You use the **receive** method to consume a message **synchronously**
- To consume a message **asynchronously**, you can register a **MessageListener** object with a specific message consumer by using the *setMessageListener* method

```
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```

- The JMS provider automatically calls the message listener's **`onMessage`** method whenever a message is delivered

JMS API Programming Model – Messages

- A JMS message can have three parts: an header (required), properties, and a body.
- **Message header**: contains a number of predefined fields used by both clients and providers to identify and route messages
- **Message properties**: containes additional user-defined properties
- **Message body**: the JMS API defines six different types of messages. Each message type corresponds to a different message body
  - TextMessage, MapMessage, BytesMessage, StreamMessage, etc.

JMS API Programming Model – Managing Messages

- The JMS API provides methods for creating messages of each type and for filling in their contents

```
TextMessage message =
context.createTextMessage();
message.setText(msg_text); // msg_text is a
String context.createProducer().send(message);
```

Using JMS in Java EE Applications

- JEE provides a set of annotations to easily work with JMS
- As usual, you can use resource/dependency injection for any JMS object, connection factories, sessions, JMXContext, destinations, etc.
- You can use **message-driven beans** to receive messages **asynchronously** (it usually implements the **MessageListener** interface)

Message-Driven Beans

- Using Message-driven beans you don’t have to you create a JMSContext, then create a JMSConsumer, then call setMessageListener to activate the listener
- You only need to define the class and annotate it, and the EJB container creates it for you
- The messages can be sent by any JMS application (not only JEE components)

How to implement a message-driven bean

- It must be annotated with the @**MessageDriven** annotation
- It is recommended that a message-driven bean class implement the **MessageListener** interface, which means that it must provide an **onMessage** method

```
void onMessage (Message inMessage)
```

- The **onMessage** method contains the business logic that handles the processing of the message
- It typically contains an **activationConfig** element containing @**ActivationConfigProperty** annotations that specify properties used by the bean

Message-Driven bean example (1)


```
@MessageDriven(activationConfig = {
        @ActivationConfigProperty(propertyName = "destinationLookup",
                propertyValue = "jms/MyQueue"),
        @ActivationConfigProperty(propertyName = "destinationType",
                propertyValue = "javax.jms.Queue")
})
public class SimpleMessageBean implements MessageListener {

        static final Logger logger = Logger.getLogger("SimpleMessageBean");

        public SimpleMessageBean() {

        }

        @Override
        public void onMessage(Message inMessage) {
                . . .
        }

}
```

Message-Driven bean example (2)


```
@Override
public void onMessage(Message inMessage) {
          try {
                    if (inMessage instanceof TextMessage) {
                              logger.log(Level.INFO,
                              "MESSAGE BEAN: Message received: {0}",
                              inMessage.getBody(String.class)
                              );
                    }
                    else {
                              logger.log(
                              Level.WARNING,
                              "Message of wrong type: {0}",
                              inMessage.getClass().getName());
                    }
          } catch (JMSException e) {
                    logger.log(
                    Level.SEVERE,
                    "SimpleMessageBean.onMessage: JMSException: {0}",
                    e.toString());
          }
}
```

Message-Driven and Session Beans

- The most visible difference between message-driven beans and session beans is that clients do not directly access them
- Like stateless session beans, message-driven beans retain no data or conversational state for a specific client. Thus, the pooling mechanism is applied by the message-driven beans container
- The container can assign an incoming message to any message-driven bean instance in the pool
- A client accesses a message-driven bean through the JMS service by sending messages to the message destination for which the message-driven bean class is the MessageListener

Message-Driven Beans lifecycle

- The EJB container usually creates a pool of message-driven bean instances.
- If the message-driven bean uses dependency injection, the container injects these references before instantiating the instance
- If a bean implements the MessageListener interface the onMessage method is called by the bean's container when a message has arrived for the bean
- After the execution of the onMessage method the container puts back in the pool the instance of the message-driven bean

Persistence Advanced Concepts

- Recall:
  - An entity is a Java class representing a table in a relational database
  - Each instance of an entity class represents a row in the table
  - The persistent state of an entity is represented through persistent fields, which correspond to columns in the table
  - To define an entity use the javax.persistence.Entity annotation

Using Collections in Entity Fields

- Collection-valued persistent fields must use the supported Java collection interfaces:
  - java.util.Collection
  - java.util.Set
  - java.util.List
  - java.util.Map
  - …
- The parametric type of a collection can be either a basic type (e.g. Integer, String) or a user-defined class
- A field consisting of a collection must be annotated with the *javax.persistence.ElementCollection* annotation

Collections in entity field: esample

- @Entity
  public class Person {
  …
  @ElementCollection(fetch=EAGER, targetClass=mypackage.Car)
  protected Set<Car> nickname = new HashSet();
  …
  }
- The targetClass attribute specifies the class name of the type parameter
- The optional fetch attribute is used to specify whether the collection should be retrieved lazily (only when the collection is really required by the application) or eagerly (always retrieve the all collection)

Multiplicity in Entity Relationships

- Tables in a relational database are connected by relationships
- A relationship is tipically defined by a set of attributes that two tables have in common and by a multiplicity
- Given two tables A and B and a relationship between them, the multiplity defines how many rows from table B are “in a relationship” with a single row from table A (and viceversa)
- JEE allow to set multiplicity in an entity relationship by mean of annotations:
  - *javax.persistence.{OneToOne , OneToMany, ManyToOne, ManyToMany}*

Introduction to Web Services

- A web service is a software component provided through a web-accessible endpoint
- The service client and server communicate using messages in the form of self-containing documents (XML, JSON) that make very few assumptions about the technological capabilities of the receiver.
- Web services provide a standard means of interoperating between highly-decoupled software applications running on a variety of platforms and frameworks

- Web services can be implemented according to different approaches:
  - “big” Web Services
  - ”RESTful” Web Services
- JEE provides support for both “big” and “RESTful” Web Services through the JAX-WS and JAX-RS APIs
- It’s out of the scope of this course to deal with Web Services and Service Oriented Architectures, but you might have a look at this frameworks to realize your applications!