9 Structura unui mesaj JMS (*Java Message Service*)

Vom dedica acest capitol prezentării structurii interne a unui mesaj JMS, respectiv a părților interne care compun mesajul (***Figura 9.1***):

- componentele de antet (*headers*);
- atributele mesajului (*properties*);
- diverse tipuri de date utile manipulate de aplicație (*message payloads*).

Clasa **`Message`** reprezintă cea mai importantă parte din specificațiile care stau la baza JMS. Toate datele și evenimente utilizate de o aplicație fundamentată pe JMS sunt comunicate prin mesaje, celelalte componente ale interfeței JMS au menirea de facilita transferul acestor mesajelor.

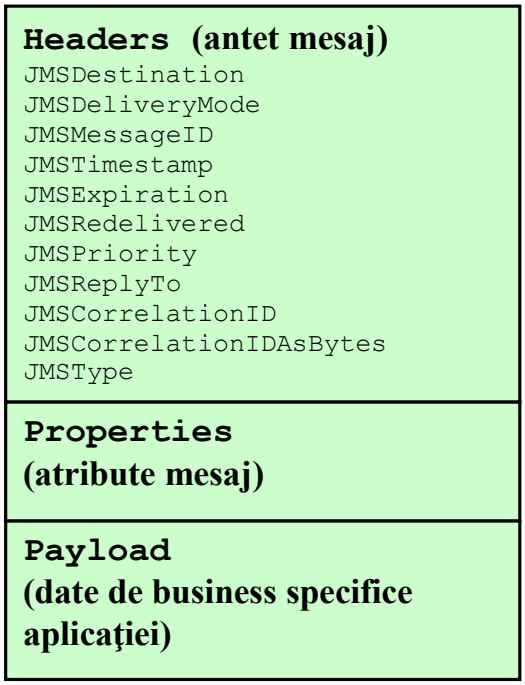

Figura 9.1 – Structura unui mesaj JMS

Un mesaj JMS joacă un rol unic domeniul sistemelor distribuite. El poate transporta atât date specifice de aplicație, cât și notificări pentru evenimente legate de activitatea aplicațiilor sistemului. În sistemele bazate pe RPC (CORBA, Java RMI, DCOM), un mesaj reprezintă o comandă de a executa o metodă sau o procedură, mecanism care blochează apelantul până la primirea unui răspuns. Un mesaj JMS nu este o comandă, prin el se transferă date și se poate informa aplicația care l-a recepționat că a avut loc un anumit eveniment. Un astfel de mesaj nu îi impune aplicației care îl recepționează ce anume să facă cu el, iar aplicația care l-a produs nu așteaptă după un răspuns. Se obține în acest fel o decuplare a producătorului mesajului de consumatorul acestuia, făcând

sistemele orientate pe mesaje mult mai dinamice și mai flexibile decât cele bazate pe paradigma cerere-răspuns realizate în manieră sincronă.
Mesajele JMS pot fi de mai multe tipuri, în funcție de maniera în care sunt organizate datele utile de transferat (*payload*), respectiv pot fi foarte structurate, cum sunt **`StreamMessage`** și **`BytesMessage`**, sau aproape deloc structurate, cum sunt tipurile **`TextMessage`**, **`ObjectMessage`** și **`MapMessage`**. Așa cum am menționat mai sus, în toate cazurile, mesajele pot transporta atât notificări cât și date de business.

9.1 Antetul unui mesaj JMS (*Headers*)

Antetul unui mesaj JMS conține date globale ce descriu cine sau ce aplicație a creat mesajul, când a fost acesta creat, cât timp urmează să fie valide datele de business pe care le conține etc. Antetul conține, de asemenea, informații de rutare a mesajului, care descriu destinația acestuia (*topic* sau *queue*), cum trebuie să se realizeze confirmarea de primire a acestuia ș.a.
Fiecare mesaj JMS conține un set de date standard în antet. Fiecare dintre acestea este accesată printr-un set de metode care urmează o sintaxă similară:

- **`setJMS<HEADER>()`**
- **`getJMS<HEADER>()`**

Prezentăm în continuare toate aceste metode de acces la componentele antetului unui mesaj JMS, ele făcând parte din descrierea interfeței **`Message`**:

```
package javax.jms;

public interface Message {

    public Destination getJMSDestination() throws JMSException;
    public void setJMSDestination(Destination destination)
        throws JMSException;

    public int getJMSDeliveryMode() throws JMSException;
    public void setJMSDeliveryMode(int deliveryMode)
        throws JMSException;

    public String getJMSMessageID() throws JMSException;
    public void setJMSMessageID(String id) throws JMSException;

    public long getJMSTimestamp() throws JMSException;
    public void setJMSTimestamp(long timestamp) throws JMSException;

    public long getJMSExpiration()throws JMSException;
    public void setJMSExpiration(long expiration) throws JMSException;

    public boolean getJMSRedelivered() throws JMSException;
```

```
    public void setJMSRedelivered(boolean redelivered)
        throws JMSException;

    public int getJMSPriority() throws JMSException;
    public void setJMSPriority(int priority) throws JMSException;

    public Destination getJMSReplyTo() throws JMSException;
    public void setJMSReplyTo(Destination replyTo) throws JMSException;

    public String getJMSCorrelationID() throws JMSException;
    public void setJMSCorrelationID(String correlationID)
        throws JMSException;

    public byte[] getJMSCorrelationIDAsBytes() throws JMSException;
    public void setJMSCorrelationIDAsBytes(byte[] correlationID)
        throws JMSException;

    public String getJMSType() throws JMSException;
    public void setJMSType(String type) throws JMSException;
}
```

Elementele de antet JMS sunt grupate în două categorii: elemente asignate automat și elemente asignate de programator.

9.2 Elementele de antet asignate automat

Majoritatea antetelor mesajelor JMS sunt asignate automat. Valorile elementelor acestor antete sunt setate de serverul de mesagerie atunci când mesajul este livrat, astfel încât valorile asignate de programator prin utilizarea metodelor **`setJMS<HEADER>()`** sunt ignorate. Cu alte cuvinte, pentru majoritatea antetelor, utilizarea metodelor de setare explicită este superfluă. Cu toate acestea, unele elemente de antet asignate automat pot fi setate în mod programatic la instanțierea claselor **`Session`** și **`MessageProducer`** (respectiv **`TopicPublisher`**). În această situație se găsesc elementele de antet **`JMSDeliveryMode`** și **`JMSPriority`**, așa cum vom vedea în continuare.

`JMSDestination`

Prin acest element de antet se identifică destinația mesajului, care poate fi fie un obiect de tip **`Topic`** fie unul de tip **`Queue`**, ambele derivate din tipul **`Destination`**. Identificarea destinației mesajului este importantă, mai ales pentru clienții care consumă mesaje de la mai mult de un *topic* sau *queue*:

```
Topic destination = (Topic) message.getJMSDestination();
```

JMSDeliveryMode

JMS oferă două moduri de livrarea mesajelor: persistent și nepersistent. Un mesaj persistent este livrat o dată și numai o singură dată, ceea ce înseamnă că dacă serverul de mesagerie înregistrează o disfuncționalitate mesajul nu este pierdut; el va fi livrat după ce serverul își reface starea. Un mesaj nepersistent este livrat cel mult o dată, ceea ce implică faptul că el poate fi pierdut pentru totdeauna, dacă serverul de mesaje eșuează în funcționare. În ambele moduri de livrare, persistent și nepersistent, serverul de mesagerie nu trebuie să trimită un mesaj către un același consumator mai mult de o dată (lucru posibil totuși prin utilizarea **JMSRedelivered**).

```
int deliveryMode = message.getJMSDeliveryMode();
if (deliveryMode == javax.jms.DeliveryMode.PERSISTENT) {
  ...
} else { // înseamnă DeliveryMode.NON_PERSISTENT
  ...
}
```

Modul de livrare al unui mesaj poate fi setat utilizând **setJMSDeliveryMode(int deliveryMode)** la momentul când mesajul este produs (**TopicPublisher** sau **QueueSender)**. Odată ce modul de livrare este setat la nivel clasei **MessageProducer**, el este aplicat tuturor mesajelor livrate de respectivul producător. Valoarea implicită este **PERSISTENT**.

```
// setarea modului de livrare JMS la nivelul producătorului
TopicPublisher topicPublisher = TopicSession.createPublisher(topic);
topicPublisher.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

JMSMessageID

Acest element de antet este un **String** a cărui valoare identifică în mod unic un mesaj în cadrul sistemului de mesagerie. **JMSMessageID** poate fi foarte util atunci când un consumator dorește să arhiveze toate mesajele pe care le-a primit și prelucrat, iar aceste mesaje trebuie să fie indexate de o manieră unică. Utilizat în conexiune cu **JMSCorrelationID**, **JMSMessageID** este de asemenea util pentru a pune în corespondență mesajele de cerere cu cele de răspuns, în cazul unei comunicări asincrone de tip cerere-răspuns.

```
String messageID = message.getJMSMessageID();
```

JMSTimestamp

Acesta este în mod automat setat de către producătorul mesajului în momentul când invocă metoda **send()**. **JMSTimestamp** conține timpul la care mesajul a fost recepționat de serverul de mesagerie și nu timpul la care a fost de fapt livrat de producător. Acest element de antet este util pentru determinarea duratei timp scurse de la momentul la care mesajul a fost trimis către server și momentul la care mesajul a fost consumat. Este reprezentat printr-o valoare de tip **long**, care păstrează timpul în milisecunde (scurse de la 1 ianuarie 1970).

```
long timestamp = message.getJMSTimestamp();
```

JMSExpiration

Prin setarea momentului la care un mesaj este considerat expirat se poate previne livrarea către consumatori a unui mesaj care nu mai este de actualitate, din perspectiva informației ce o poartă. Este util pentru mesaje ale căror componentă de date de business este validă pentru o perioadă limitată de timp.

```
long timeToLive = message.getJMSExpiration();
```

Timpul până la expirarea mesajului este setat în milisecunde de către producător (de exemplu **`TopicPublisher`**) utilizând metoda **`setTimeToLive()`**.

```
TopicPublisher topicPublisher = TopicSession.createPublisher(topic);
// setare timp de viaţă la 1 oră (1000 milisecunde x 60 secunde x 60 minute)
topicPublisher.setTimeToLive(3600000);
```

Serverul de mesagerie va adăuga valoarea **`timeToLive`** la timpul sistemului şi va seta **`JMSExpiration`**. Implicit, **`timeToLive`** este zero, ceea ce indică faptul că mesajul nu va expira. Prin apelul metodei **`setTimeToLive()`** cu argumentul zero (0) se asigură faptul că mesajul este creat fără a avea o dată de expirare. Orice apel programatic direct al metodei **`setJMSExpiration()`** va fi ignorat la momentul la care mesajul este trimis.

`JMSRedelivered`

Acest element de antet indică faptul că mesajul se dorește a fi retrimis către consumator. Este modelat ca o valoare booleană, când **`JMSRedelivered`** este *true* atunci mesajus este retransmis, când este *false* nu se realizează acest lucru. Un mesaj poate fi marcat pentru retransmitere atunci când consumatorul a eșuat să confirme primirea acestuia la livrarea anterioară, sau când serverul de mesagerie nu este sigur dacă cel ce este de presupus să consume mesajul l-a primit deja sau nu.

```
boolean isRedelivered = message.getJMSRedelivered();
```

`JMSPriority`

Serverul de mesagerie poate asigna o anumită prioritate unui mesaj atunci când acesta este transmis în sistem. Există două categorii de priorități asociate mesajelor:

- nivelurile de la 0 la 4 modulează gradual o prioritate normală;
- nivelurile de la 5 la 9 modulează gradual o prioritate de rangul urgență/alertă (*expedited*).

Serverul de mesagerie poate utiliza prioritatea mesajului pentru a da întâietate unor mesaje care trebuie să ajungă mai rapid la consumatori. Mesajele din categoria urgență/alertă au întâietate față de cele cu o prioritate normală.

```
int priority = message.getJMSPriority();
```

Nivelul de prioritate asociat unui mesaj poate fi declarat de către un client JMS prin utilizarea metodei **`setPriority()`** la nivel producătorului mesajului.

```
TopicPublisher topicPublisher = TopicSession.createPublisher(topic);
topicPublisher.setPriority(9);
```

Orice apel programatic direct al metodei **`setJMSPriority()`** va fi ignorat la momentul transmiterii mesajului.

9.3 Elementele antetului asignabile de către programator

Chiar dacă cele mai multe elemente ale antetului unui mesaj JMS sunt asignate automat de către serverul de mesagerie la momentul transmiterii mesajului, unele pot fi totuși setate explicit la nivelul obiectului **`Message`** înainte ca acesta să fie livrat de către producător.

`JMSReplyTo`
În unele cazuri, producătorul unui mesaj JMS poate dori ca aplicația care va consuma mesajul să trimită înapoi un mesaj de răspuns (de exemplu atunci când se folosește un mecanism de cerere-răspuns prin intermediul platformei de mesagerie). Elementul de antet **`JMSReplyTo`**, care conține un obiect de tip **`javax.jms.Destination`**, indică care este adresa la care consumatorul JMS va trebui să trimită răspunsul. Utilizarea acestei proprietăți de la nivelul antetului, oferă posibilitate unei decuplări efective a producătorului mesajului de consumatorul acestuia, în cadrul scenariului de cerere-răspuns. Trebuie remarcat faptul că aplicația care joacă rolul de consumator al mesajului nu trebuie să trimită un răspuns numai pentru că **`JMSReplyTo`** este setat.

```
message.setReplyTo(topic);
...
Topic topic =  (Topic) message.getJMSReplyTo();
```

`JMSCorrelationID`
Acest element al antetului oferă posibilitatea asocierii mesajului curent cu un alt mesaj transmis anterior, sau cu un identificator specific de aplicație. Cel mai adesea, **`JMSCorrelationID`** este utilizat pentru a marca un mesaj ca răspuns la un mesaj trimis anterior și identificat prin **`JMSMessageID`**.
Totuși, trebuie reținut că **`JMSCorrelationID`** poate lua orice valoare, nu numai **`JMSMessageID`**.

```
message.setJMSCorrelationID(identificator);
...
String correlationID = message.getJMSCorrelationID();
```

`JMSType`
Este un element opțional al antetului, care este setat de clientul JMS. Menirea lui este aceea de a identifica structura mesajului și tipul de date de business. Trebuie remarcat faptul că acest element al antetului nu indică ce clasă de mesaj este trimisă (**`MapMessage`**, **`TextMessage`**, **`ObjectMessage`** etc.) ci, mai curând, un tip de înregistrare ce este folosit de componenta de persistare a serverului de mesagerie. Unele MOM, cum ar fi de exemplu WebSphere MQ creat de IBM, tratează corpul mesajului ca un șir neinterpretat de octeți. Astfel de sisteme folosesc adesea acest tip de mesaj ca o metodă simplă, pusă la dispoziția aplicațiilor, de a eticheta corpul unui mesaj ca fiind unul de tip JMS. În consecință, tipul de mesaj pate fi util atunci când se interschimbă

mesaje cu clienți non-JMS, care au nevoie de această informație pentru a putea prelucra corpul, conținând datele de business ale mesajului.

9.4 Atributele unui mesaj JMS

Atributele unui mesaj JMS acționează ca elemente de caracterizare suplimentară a mesajului, pe lângă cele conținute de antet. Ele permit programatorului să adauge mai multe informații structurate cu privire la conținutul mesajului. Sunt, de asemenea, utilizate pentru a expune date de identificare ce pot fi ulterior prelucrate de componentele de selectare a mesajelor, atunci când se dorește filtrarea acestora. Interfața **`Message`** oferă mai multe metode de acces și modificare a acestor atribute. Valoare luată de un atribut poate fi de următoarele tipuri: **`String`**, **`boolean`**, **`byte`**, **`short`**, **`int`**, **`long`**, **`float`** sau **`double`**.
Există trei categorii de atribute care se pot asocia unui mesaj JMS:

- atribute specifice aplicației;
- atribute definite de către interfața JMS;
- atribute specifice serverului de mesagerie.

9.4.1 Atribute specifice aplicației

Orice atribut (proprietate) definit de programatorul de aplicație poate fi considerat un atribut specific aplicației. Atributele specifice aplicației sunt setate înainte ca mesajul să fie livrat sistemului. Ne există atribute predefinite; programatorul are libertate deplină în a-și defini atributele care consideră că pot fi utile în prelucrarea mesajului, din perspectiva logicii mai ample a unui sistem informatic distribuit realizat pe baza unei platforme de mesagerie. Așa cum vom vedea în capitolele următoare, unul din atributele uzual folosite de programatori este acela prin care specifică numele utilizatorului (sau aplicației) care trimite un mesaj. Prezentăm în continuare subsetul de metode ce fac parte din interfața **`Message`** și care sunt destinate accesului și setării diferitelor tipuri de atribute.

```
package javax.jms;


public interface Message {

    public String getStringProperty(String name)
        throws JMSException, MessageFormatException;
    public void setStringProperty(String name, String value)
        throws JMSException, MessageNotWriteableException;


    public int getIntProperty(String name) throws JMSException, MessageFormatException;

    public void setIntProperty(String name, int value) throws JMSException,
        MessageNotWriteableException;

    public boolean getBooleanProperty(String name)
        throws JMSException, MessageFormatException;
    public void setBooleanProperty(String name, boolean value)
        throws JMSException, MessageNotWriteableException;
```

```
    public double getDoubleProperty(String name)
        throws JMSException, MessageFormatException;
    public void setDoubleProperty(String name, double value)
        throws JMSException, MessageNotWriteableException;


    public float getFloatProperty(String name)
        throws JMSException, MessageFormatException;
    public void setFloatProperty(String name, float value)
        throws JMSException, MessageNotWriteableException;


    public byte getByteProperty(String name) throws JMSException,
        MessageFormatException;

    public void setByteProperty(String name, byte value) throws JMSException,
        MessageNotWriteableException;



    public long getLongProperty(String name) throws JMSException,
        MessageFormatException;

    public void setLongProperty(String name, long value) throws JMSException,
        MessageNotWriteableException;



    public short getShortProperty(String name) throws JMSException,
        MessageFormatException;
    public void setShortProperty(String name, short value)
        throws JMSException, MessageNotWriteableException;


    public Object getObjectProperty(String name) throws JMSException,
        MessageFormatException;
    public void setObjectProperty(String name, Object value)
        throws JMSException, MessageNotWriteableException;

    public void clearProperties() throws JMSException;

    public Enumeration getPropertiyNames() throws JMSException;

    public boolean propertyExists(String name) throws JMSException;

    ...
}
```

De remarcat faptul că metodele pentru lucru cu atribute de tip **`Object`** (**`getObjectProperty()`** și **`setObjectProperty()`**) pot fi utilizate numai cu obiecte care corespund tipurilor admise și anume tipurile primitive (**`java.lang.Integer`**, **`java.lang.Double`** etc.) și tipul **`String`**. Ele nu pot fi utilizate cu alte obiecte Java, cum ar fi cele ce modelează aspecte complexe de business.

Odată ce un mesaj este publicat sau transmis, proprietățile asociate acestuia devin protejate la scriere (*read-only*) și nu mai pot fi schimbate nici de către consumator, nici de către producător.
Dacă consumatorul încearcă să seteze un atribut, metoda invocată pentru aceasta va genera o excepție de tipul **`javax.jms.MessageNotWriteableException`**.
Atributele pot fi totuși schimbate la nivel de mesaj prin invocarea metodei **`clearProperties()`**, care va șterge toate atributele mesajului, permițând astfel adăugarea unor atribute noi.
Metoda **`getPropertyNames()`**, componentă a interfeței **`Message`**, poate fi utilizată pentru a obține într-un tip **`Enumeration`** toate numele atributelor conținute de un mesaj. Acest nume pot fi apoi utilizate pentru extragerea valorilor atributelor în cauză, utilizând metodele de acces enumerate mai sus pentru diferitele tipuri de date.

```
public void onMessage(Message message) {
    Enumeration propertyNames = message.getPropertyNames();
    while(propertyNames.hasMoreElements() {
        String name = (String) propertyNames.nextElement();
        Object value = getObjectProperty(name);
        System.out.println("\nName = " + name + ", Value = " + value);
    }
}
```

9.4.2 Atribute definite de către interfața JMS

Atributele definite de către interfața JMS prezintă aceleași caracteristici ca și cele specifice aplicației, cu excepția faptului că cele mai multe dintre ele sunt setate de serverul de mesagerie, la momentul transmiterii unui mesaj. Atributele definite de interfața JMS acționează în fapt ca un set opțional de atribute setate de serverul de mesagerie, pe lângă cele setate în partea de antet al mesajului. Prezentăm mai jos lista celor nouă atribute definte de JMS:

- `JMSXUserID`
- `JMSXAppID`
- `JMSXProducerTXID`
- `JMSXConsumerTXID`
- `JMSXRcvTimestamp`
- `JMSXDeliveryCount`
- `JMSXState`
- `JMSXGroupID`
- `JMSXGroupSeq`

Producătorii de platforme de mesagerie care respectă cerințele interfeței JMS, pot alege să ofere suport pentru toate aceste atribute sau numai pentru unele dintre ele. Cu toate acestea, tuturor implementărilor comerciale de JMS li se cere în mod obligatoriu să ofere suport pentru următoarele două atribute: **`JMSXGroupID`** și **`JMSXGroupSeq`**. Aceste două atribute opționale sunt utilizate la gruparea mesajelor. Trebuie remarcat că în interfața **`Message`** nu se vor regăsi, ca în cazul atributelor din antet, metode specifice definite într-un format de tipul **`getJMSX<PROPERTY>`** și **`setJMSX<PROPERTY>`**. Aceste atribute

opționale, atunci când sunt utilizate, trebuie să fie setate în aceeași manieră ca atributele specifice aplicației.

```
message.setStringProperty("JMSXGroupID", "OMS-001");
message.setIntProperty("JMSXGroupSeq", 11);
```

9.4.3 Atribute specifice serverului de mesagerie

Fiecare furnizor de *middleware* orientat pe mesaje poate să-și definească o colecție proprie și specifică de atribute, care pot fi setate fie de către client, fie de către server în mod automat. Aceste atribute specifice produsului de mesagerie trebuie să fie prefixate de textul **`JMS`**, urmat de numele proprietății (**`JMS_<vendor-property-name>`**). Scopul acestor atribute specifice furnizorului de mesagerie este acela de a oferi suport pentru diversele facilități furnizate în mod particular de un produs sau altul.

10 Tipuri de mesaje JMS

Java Mesage Service definește șase tipuri de interfețe **`Message`** care trebuie să fie oferite de furnizorii de mesagerie compatibilă JMS. Implementare acestor interfețe este însă lăsată la latitudinea producătorilor. Cele șase interfețe sunt formate din interfața **`Message`** și cinci subinterfețe ale sale:

- **`TextMessage`**
- **`StreamMessage`**
- **`MapMessage`**
- **`ObjectMessage`**
- **`BytesMessage`**

`Message`
Tipul cel mai simplu de mesaj JMS este modelat de **`javax.jms.Message`** și servește și ca interfață de bază pentru celelalte interfețe. Tipul **`Message`** poate fi creat și folosit ca mesaj JMS și fără a purta conținut de business (*payload*).

```
// crează și transmite un Message
Message message = session.createMessage();
publisher.publish(message);

...

// recepționarea unui Message la consumator
public void onMessage(Message message) {
    // fără date de business, se procesează doar ca o notificare a unui eveniment
}
```

Acest tip de mesaj conține numai antetul și atributele (proprietățile) JMS, fiind utilizat pentru notificarea de evenimente. Notificarea unui eveniment care a avut loc în sistem poate îmbrăca forma unui *broadcast*, a unei avertizări, sau a unei note informative cu

privire la evenimentul care a avut loc. În astfel de scenarii, utilizarea unui obiect de tip **Message**, fără încărcătură de business, este cea mai eficientă modalitate de comunicare între aplicațiile sistemului.

TextMessage

Acest tip de mesaj transportă un obiect de tip **java.lang.Strig** ca încărcătură. Este util atunci când este necesar schimbul de mesaje simple de tip text, sau când se utilizează structuri de date mai complexe, cum ar fi documente XML.

```
public interface TextMessage extends Message {
    public String getText() throws JMSException;
    public void setText(String payload) throws JMSException,
        MessageNotWriteableException
}
```

Mesajele de tip text pot fi create folosind una din cele două metodele de tip *factory*, definite la nivelul interfeței **Session**. Una dintre ele nu preia nici un argument, ceea ce implică crearea unui obiect de tip **TextMessage** vid, fără încărcătură de business, urmând ca aceasta să fie adăugată ulterior, prin invocarea metodei **setText()**, ce aparține interfeței **TextMessage**. A două metodă oferită pentru crearea unui mesaj de tip **TextMessage** preia ca argument o încărcătură de tip **String**, dând astfel posibilitatea creării unui mesaj gata de a fi transmis.

```
TextMessage textMessage = session.createTextMessage();
textMessage.setText(”un mesaj de tip text”);
topicPublisher.publish(textMessage);
...

TextMessage textMessage = session.createTextMessage(”un mesaj de tip text”);
queueSender.send(textMessage);
...
```

Când consumatorul primește un mesaj de tip **TexMessage**, încărcătura acestuia poate fie extrasă într-un **String** folosind metoda **getText()**. Dacă mesajul de tip **TextMessage** este folosit fără date utile, atunci **getText()** va returna valoare **null**.

ObjectMessage

Acest tip de mesaj este destinat transportului de obiecte Java serializabile. Este foarte util atunci când se interschimbă date între aplicații scrise în Java.

```
public interface ObjectMessage extends Message {
    public java.io.Serializable getObject() throws JMSException;
    public void setObject(java.io.Serializable payload) throws JMSException,
        MessageNotWriteableException
}
```

Mesajele de acest tip pot fi create cu una din cele două metode oferite de interfața **Session**. Similar cu modelul prezentat pentru timpul de mesaj **TextMessage**, una dintre metode nu preia nici un argument ca *payload*, acesta putând fi adăugat ulterior prin apelul metodei **setObject()**, la nivelul interfeței **ObjectMessage**. Cealaltă metodă

preia ca argument o valoare de tip **`Serializable`**, producând un mesaj de tip **`ObjectMessage`** gata de a fi transmis.

```
// Order este un obiect serializabil
Order order = new Order();
...

ObjectMessage objectMessage = session.createObjectMessage();
ojectMessage.setObject(order);
queueSender.send(objectMessage);
...

ObjectMessage objectMessage = session.createObjectMessage(order);
topicPublisher.publish(objectMessage);
...
```

Atunci când un consumator primește un mesaj de tip **`ObjectMessage`**, poate extrage conponenta de date de business utilizând metoda **`getObject()`**. Dacă un mesaj **`ObjectMessage`** este transmis fără încărcătura de date utile, atunci metoda **`getObject()`** returnează **`null`**.

```
public void onMessage(Message message) {
    try {
        ObjectMessage objectMessage = (ObjectMessage)message;
        Order order = (Order)objectMessage.getObject();
        ...
    } catch (JMSException jms) {
        ...
    }
}
```

Trebuie menționat faptul că, chiar dacă acest tip de mesaj este foarte convenabil pentru schimbul de date între aplicații Java, aceasta poate fi o cerință foarte dificil de satisfăcut atunci când avem de a face cu medii vechi, eterogene, în care rulează aplicații scrise pentru diverse platforme. De asemenea, trebuie avut în vedere faptul că definițiile claselor obiectelor care sunt utilizate în tranzacții trebuie să fie disponibile atât producătorului cât și consumatorului JMS. În cazul unui sistem nou, proiectat în mod programatic pentru platforma Java, acest tip de mesaj oferă în schimb simplitate și claritate codului.

`BytesMessage`

Acest tip de mesaj transportă, ca date utile, un masiv octeți. Este util pentru comunicarea de date între aplicații în format nativ, atunci când alte tipuri de mesaje nu sunt compatibile. De asemenea, este util atunci când JMS este utilizat în mod esențial ca un mijloc de transport între două sisteme și încărcătura de date este transparentă pentru clientul JMS.

```
public interface BytesMessage extends Message {
    public long getBodylength() throws JMSException;
    public byte readByte() throws JMSException;
```

```
    public void writeByte(byte value) throws MSException;
    public int readUnsignedByte() throws MSException;

    public int readBytes(byte[] value) throws JMSException;
    public void writeBytes(byte[] value) throws JMSException;
    public int readBytes(byte[] value, int length) throws JMSException;
    public void writeBytes(byte[] value, int offset, int length) throws JMSException;

    public boolean readBoolean() throws JMSException;
    public void writeBoolean(boolean value) throws )MSException;


    public char readChar() throws JMSException;


    public void writeChar(char value) throws JMSException;


    public short readShort() throws JMSException;


    public void writeShort(short value) throws JMSException;
    public int readUnsignedShort() throws JMSException;

    public void writeInt(int value) throws JMSException;
    public int readInt() throws JMSException;

    public void writeLong(long value) throws JMSException;
    public long readLong() throws JMSException;

    public float readFloat() throws JMSException;
    public void writeFloat(float value) throws JMSException;

    public double readDouble() throws JMSException;
    public void writeDouble(double value) throws JMSException;


    public String readUTF() throws JMSException;
    public void writeUTF(String value) throws JMSException;

    public void writeObject(Object value) throws JMSException;

    public void reset() throws JMSException;

}
```

Cele mai multe dintre metodele definite în interfața **`BytesMessage`** îi permit programatorului să citească date dinspre și să scrie către un șir de *bytes*, folosind tipuri primitive ale limbajului Java. Când unul din tipurile primitive din Java este scris într-un șir de *bytes*, utilizând una din metodele de tipul **`write<TYPE>()`**, valoarea primitivei este convertită către reprezentarea ei în tipul **`byte`** și este apoi adăugată fluxului de date de transmis.

```
BytesMessage bytesMessage = session.createBytesMessage();

bytesMessage.writeChar(`A`);
bytesMessage.writeInt(2009);
bytesMessage.writeUTF(”ASE”);
...
queueSender.send(bytesMessage);
```

Când un astfel de mesaj este recepționat de un consumator, datele de business se găsesc sub forma unui flux de octeți, deci este practic posibil să fie citite într-o manieră arbitrară, însă acest lucru ar determina o interpretare eronată a acestor date. Ceea ce trebuie făcut în momentul consumului acestui flux de octeți este citirea acestuia în aceeași ordine în care a fost creat și folosind metodele corespondente de conversie către primitivele Java.

```
public void onMessage(Message message) {
    try {
        BytesMessage bytesMessage = (BytesMessage)message;
        char c = bytesMessage.readChar();
        int I = bytesMessage.readInt();
        String s = bytesMessage.readUTF();
    } catch (JMSException jms) {
        ...
    }
}
```

Pentru a citi și scrie valori de tip **`String`**, interfața **`BytesMessage`** utilizează metode bazate pe formatul UTF-8, care este un format standard pentru transferul și stocarea eficientă de text Unicode.
Pe lângă metodele pentru accesarea datelor de bază, interfața **`BytesMessage`** mai include și o singură metodă **`writeObject()`**. Aceasta este utilizată pentru obiecte de tip **`String`** și pentru obiectele care emulează tipurile de bază: **`Byte`**, **`Boolean`**, **`Character`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`**, **`Double`**. Când sunt scrise către un obiect de tip **`BytesMessage`**, aceste valori sunt convertite la formatul de tip **`byte`** corespunzător primitivei. Metoda **`writeObject()`** este furnizată ca o modalitate comodă de scrie date ale căror tip nu este cunoscut până la momentul execuției.
Metoda **`reset()`** returnează un pointer către începutul fluxului de *bytes* și pune obiectul de tip **`BytesMessage`** în modul de acces numai pentru citire (*read-only*), astfel încât conținutul lui să nu mai poată fi modificat. Această metodă poate fi apelată în mod explicit de către clientul JMS atunci când, spre exemplu, este necesară refacerea unei operațiuni de citire din fluxul de *bytes*, în urma unei erori de citire. Ea este întotdeauna apelată implicit atunci când mesajul de tip **`BytesMessage`** este livrat.
Tipul de mesaj **`BytesMessage`** este unul dintre cele mai portabile soluții atunci când comunicarea este necesar a fi realizată cu clienți non-JMS. În unele cazuri, un client JMS poate să joace rolul unui *router*, consumând mesaje de la o anumită sursă și trimițându-le către o altă destinație. Într-o astfel de situație, aplicația de rutare nu este interesată de conținutul datelor transferate și poate prefera transferul acestora în format binar.

`StreamMessage`

Acest tip de interfață este destinat transportului unui flux de primitive Java (**`int`**, **`double`**, **`char`** etc.). Interfața oferă o colecție de metode care asigură punerea în corespondență a unui flux formatat de octeți cu primitivele Java. Tipurile de bază Java sunt citite din mesaj în aceeași ordine în care au fost scrise. La prima vedere, interfața **StreamMessage** se aseamănă foarte mult cu interfața **BytesMessage**, însă ele nu sunt deloc la fel.

```
public interface StreamMessage extends Message {

   public boolean readBoolean() throws JMSException;
   public void writeBoolean(boolean value) throws JMSException;

   public byte readByte() throws JMSException;
   public int readBytes(byte[] value) throws JMSException;
   public void writeByte(byte value) throws JMSException;
   public void writeBytes(byte[] value) throws JMSException;
   public void writeBytes(byte[] value, int offset, int length)
       throws JMSException;

   public short readShort() throws JMSException;
   public void writeShort(short value) throws JMSException;

   public char readChar() throws JMSException;
   public void writeChar(char value) throws JMSException;

   public int readInt() throws JMSException;
   public void writeInt(int value) throws JMSException;

   public long readLong() throws JMSException;
   public void writeLong(long value) throws JMSException;

   public float readFloat() throws JMSException;
   public void writeFloat(float value) throws JMSException;

   public double readDouble() throws JMSException;
   public void writeDouble(double value) throws JMSException;

   public String readString() throws JMSException;
   public void writeString(String value) throws JMSException;

   public Object readObject() throws JMSException;
   public void writeObject(Object value) throws JMSException;

   public void reset() throws JMSException;
}
```

Interfața **StreamMessage** urmărește ordinea și tipul primitivelor care sunt scrise în fluxul de octeți aplicându-se, în consecință, reguli de conversie. De exemplu, încercarea de a citi o primitivă ca și **long**, dacă în fluxul de octeți a fost scrisă de tip **short**, va genera o excepție. Acest fapt nu se întâmplă în cazul interfeței **BytesMessage**. În cadrul interfeței **BytesMessage** se poate scrie un **long** pe 64 *bits* (8 octeți), ca un șir nestructurat de octeți și poate fi citită ulterior o porțiune din acest flux într-un **short** pe 16 *bits* (practic se citesc primii 2 octeți ai valorii **long**). Pe de altă parte, în cadrul interfeței **StreamMessage**, pe lângă valoarea **long** a primitivei, este scrisă și informația cu privire la tipul de date, impunându-se un set strict de reguli de conversie la momentul citirii, pentru a se preveni citirea unui **long** ca și **short**. În tabloul de mai jos sunt prezentate regulile de conversie pentru fiecare tip de date. Coloana din stânga conține tipul de date la scriere, în timp ce coloana din dreapta arată cum poate fi citit tipul respectiv de date. În cazul în care se încearcă citirea unei date de tip **long** ca și **short**, atunci interfața generează o excepție **JMSException**, prin metoda care facilitează

accesul la date, prin care se va indica faptul că tipul original de date nu a putut fi convertit către tipul cerut.

write<TYPE>()	**read<TYPE>()**
boolean	boolean,String
byte	byte,short,int,long,String
short	short,int,long,String
char	char,String
long	long,String
int	int,long,String
float	float,double,String
double	double,String
String	String,boolean,byte,short,int,long,float,double
byte []	byte []

Se observă că valorile de tip **Strig** pot fi convertite către oricare din primitivele Java, dacă au fost formatate corect.
Dacă o valoare **String** nu poate fi convertită către tipurile primitive cerute, atunci este generată o excepție **java.lang.NumberFormatException**. Totuși, cele mai multe dintre valorile tipurilor primitive pot fi accesate ca **String**, prin utilizarea metodei **readString()**. Singurele excepții de la această regulă le constituie valorile ce tip **char** și masivele de **byte**, care nu pot fi citite ca **String**.
Metoda **writeObject()** urmează aceleași reguli ca în cazul metodei omonime din interfața **BytesMessage()**. Primitivele emulate de clasele **Byte**, **Boolean**, **Character**, **Short**, **Integer**, **Long**, **Float**, **Double** sunt convertite către tipurile de bază corespunzătoare. Metoda **readObject()** returnează fie clasa corespunzătoare valorii primitive conținute de mesaj, fie un **String** sau un masiv de **char**, depinzând de tipul de date care a fost scris la producător. De exemplu, dacă valoarea scrisă a fost o primitivă **int**, atunci poate fi citită ca un obiect **java.lang.Integer**.
Interfața **StreamMessage** permite, de asemenea, scrierea în fluxul de octeți a valorilor **null**. Atunci când un client JMS încearcă să citească o valoare **null**, utilizând metoda **readObject()**, este returnată o valoare **null** ca atare. Pentru celelalte metode de acces la primitive însă, încercarea de a converti o valoare **null** către tipul cerut, va necesita utilizarea operației **valueOf()**.
În cazul în care operația de citirea dintr-un mesaj de tip **StreamMessage** generează o excepție și nu mai este posibilă refacerea stării dinaintea citirii, fără a cere din nou mesajul sistemului de mesagerie, atunci se poate reseta pointerul asociat fluxului de octeți la poziția pe care a avut-o chiar înaintea operației de citire care a generat excepția. În felul acesta, clientul JMS are posibilitatea de a-și reface în mod programatic pointerul la fluxul de date pe care l-ar fi pierdut în cazul neprocesării excepției generate de operația nereușită de citire.
Metoda **reset()** returnează pointerul de la începutul fluxului de octeți și face mesajul accesibil numai pentru citire (*read-only*). Cu toate acestea, dacă mesajul este retransmis, va trebui să fie apelată în mod explicit de către clientul care consumă acest mesaj.

```
if (streamMessage.getJMSRedelivered())
    streamMessage.reset();
```

MapMessage

Acest tip de mesaj transportă ca date utile un set de perechi definite în forma nume-valoare. Acest tip de încărcătură este similar obiectului de tip **`java.util.Properties`**, cu excepția faptului că valorile din perechi trebuie să fie primitive Java (respectiv obiectele care le emulează), sau de tip **`String`**. Clasa **`MapMessage`** este utilă pentru transmiterea de mesaje al căror conținut (care anume perechi nume-valoare) poate fi diferit, de la un mesaj la altul. Altfel spus, această clasă oferă posibilitatea implementării unui API autodescriptiv, fiecare mesaj putând conține un subset al setului general de date care se pot interschimba între aplicații și care este definit la nivelul întregului sistemului informatic.

```
public interface MapMessage extends Message {

    public boolean getBoolean(String name) throws JMSException;
    public void setBoolean(String name, boolean value) throws JMSException;

    public byte getByte(String name) throws JMSException;
    public void setByte(String name, byte value) throws JMSException;

    public byte[] getBytes(String name) throws JMSException;
    public void setBytes(String name, byte[] value) throws JMSException;
    public void setBytes(String name, byte[] value, int offset, int length)
       throws JMSException;

    public short getShort(String name) throws JMSException;
    public void setShort(String name, short value) throws JMSException;
    public char getChar(String name) throws JMSException;
    public void setChar(String name, char value) throws JMSException;

    public int getInt(String name) throws JMSException;
    public void setInt(String name, int value)throws JMSException;

    public long getLong(String name) throws JMSException;
    public void setLong(String name, long value) throws JMSException;

    public float getFloat(String name) throws JMSException;
    public void setFloat(String name, float value) throws JMSException;

    public double getDouble(String name) throws JMSException;
    public void setDouble(String name, double value) throws JMSException;

    public String getString(String name) throws JMSException;
    public void setString(String name, String value) throws JMSException;


    public Object getObject(String name) throws JMSException;
    public void setObject(String name, Object value) throws JMSException;

    public Enumeration getMapNames( ) throws JMSException;
```

```
    public boolean itemExists(String name) throws JMSException;
}
```

În mod esențial, clasa **MapMessage** oferă o funcționalitate similară cu atributele (proprietățile) JMS prezentate anterior: orice pereche etichetă-valoare poate fi parte a încărcăturii utile a unui astfel de mesaj. Numele, sau eticheta, trebuie să fie un **String**, iar valoarea asociată poate fi, fie un **String**, fie unul din tipurile primitive Java. Valorile scrise într-un **MapMessage** pot fi citite de un consumator JMS utilizând numele ca și cheie.

```
MapMessage mapMessage = session.createMapMessage();
mapMessage.setString("userID", "OMS");
mapMessage.setString("orderID", "O200911260000011");
mapMessage.setInt("orderQuantity", 1000);
mapMessage.setFloat("orderPrice", 12.34);

...

String userID = mapMessage.getString("userID");
String orderID = mapMessage.getString("orderID");
int orderQuantity = mapMessage.getInt("orderQuantity");
float orderPrice = mapMessage.getFloat("orderPrice");
```

Metoda **setObject()** poate scrie unul din obiectele care emulează tipurile primitive, un obiect de tipul **String**, sau un masiv de octeți. La momentul setării, obiectele care emulează tipurile primitive sunt în mod automat convertite către acestea din urmă. Metoda **getObject()** poate citi date de tipul **String**, un masiv de octeți, orice primitivă, sau obiectul corespunzător care o emulează. Aceleași reguli de conversie prezentate la tipul **StreamMessage** se aplică și pentru datele manipulate de tipul **MapMessage**.
Dacă un client JMS încearcă să citească o pereche nume-valoare care nu există, atunci valoarea este tratată ca și cum ar fi fost **null**. Cu toate că metoda **getObject()** returnează **null** în cazul inexistenței perechii nume-valoare, metodele prevăzute pentru celelalte tipuri de date au un comportament diferit, după cum urmează:

- metodele pentru accesarea primitivelor generează o excepție de tipul **java.lang.NumberFormatException**, dacă este citită o valoare **null**, sau dacă se încearcă citirea unei perechi nume-valoare inexistente;
- metoda **getBoolean()** returnează **false** pentru valorile **null**;
- metoda **getString()** returnează o valoare **null** sau un **String** vid (“”), în funcție de implementare;
- metoda **getChar()** generează o excepție de tipul **NullPointerException**.

Pentru a evita citirea unei perechi nume-valoare inexistente, clasa **MapMessage** pune la dispoziția utilizatorului metoda de test **itemExists()**. Mai mult decât atât, metoda **getMapNames()** oferă posibilitatea clientului JMS să obțină o enumerare a numelor (etichetelor) folosite în mesaj, pentru ca apoi să poate citi valorile asociate lor.

```
public void onMessage(Message message) {
    MapMessage mapMessage = (MapMessage)message;
    Enumeration names = mapMessage.getNames();
    while(names.hasMoreElements()) {
```

```
        String name = (String)names.nextElement();
        Object value = mapMessage.getObject(name);
        System.out.println("\nName = " + name + ", Value = " + value);
    }
```

10.1 Confirmarea mesajelor de către client

Un consumator JMS poate să își seteze modul de confirmare a mesajelor primite la momentul la care creează o sesiune de lucru cu serverul de mesagerie. Există trei moduri de confirmare mesajelor de către un consumator JMS:

- **AUTO_ACKNOWLEDGE**
- **DUPS_OK_ACKNOWLEDGE**
- **CLIENT_ACKNOWLEDGE**

Atunci când consumatorul alege să folosească modul de confirmare **CLIENT_ACKNOWLEDGE**, acesta trebuie să folosească metoda **acknowledge()**, definită în interfața **Message**. De exemplu, pentru un consumator în cadrul modelului *pub/sub* setarea unuia din cele trei moduri de confirmare se realizează în maniera următoare:

```
TopicSession topic = topicConnection.createTopicSession(false,
    Session.CLIENT_ACKNOWLEDGE);
```

În modul **CLIENT_ACKNOWLEDGE**, consumatorului JMS trebuie în mod explicit să confirme fiecare mesaj pe care îl primește. Metoda **acknowledge()** este utilizată special pentru acest scop.

```
public void onMessage(Message message) {
    message.acknowledge();

    ...
}
```

Celelalte moduri de confirmare nu necesită utilizarea unei confirmări explicite din partea consumatorului JMS. Pentru o sesiune deschisă în mod tranzacțional, orice modalitate de confirmare specificată va fi ignorată. Când sesiunea este deschisă pentru a realiza interschimbul de mesaje în mod tranzacțional, confirmarea este parte din tranzacție și este executată automat înainte de comiterea acesteia. Dacă tranzacția este revocată, nu va fi furnizată nici o confirmare.

Specificarea modului de confirmare **DUPS_OK_ACKNOWLEDGE** la nivelul sesiunii JMS, instruiește serverul de mesagerie asupra faptului că este acceptată de către consumator trimiterea unui mesaj către acesta de mai multe ori. Această semantică este sensibil diferită față de cea cerută de modul **AUTO_ACKNOWLEDGE**, prin care serverul de mesagerie asigură transmisia unui mesaj „o dată și numai o dată” (modelul *p2p*), sau „cel mult o dată” (modelul *pub/sub*). Modelul de livrare **DUPS_OK_ACKNOWLEDGE** pleacă de la premisa că procesarea necesară pentru asigurarea unei livrări în maniera „o dată și numai o dată” implică costuri fixe mai mari, în termeni de utilizare a resurselor platformei de comunicație, aspect care ar afecta viteza de livrare a mesajelor la nivelul serverului de mesagerie. O aplicație care este tolerantă la primirea de mesaje duplicate, ar

putea folosi modul de confirmare **`DUPS_OK_ACKNOWLEDGE`** pentru a evita acest consum fix de resurse la nivelul serverului de mesagerie.
În practică însă, îmbunătățirea performanței obținute prin utilizarea modului de confirmare **`DUPS_OK_ACKNOWLEDGE`** poate fi nesemnificativă, sau chiar inexistentă, în funcție de maniera de implementare a serverului de mesagerie. În fapt, este de așteptat ca furnizorul de mesagerie JMS să ofere performanțe mai bune în modul **`AUTO_ACKNOWLEDGE`**, pentru că ar primi confirmările mai curând și nu mai târziu, iar aceasta i-ar permite eliberarea mai rapidă a resurselor ocupate, sau reducerea dimensiunilor structurilor de date stocate de o manieră persistentă și a cozilor din memorie.