CHAPTER 3

JMS Messages

All too often when people think about messaging, their minds immediately focus on the mechanics of the process and the entity for which the process is being implemented—the message—is all too easily forgotten. The message is the unit of exchange and without first defining the message, we have nothing with which to communicate. Thus, before we explore the use of the JMS API, we examine in detail the JMS message. We review the process of message definition and learn what factors influence the nature of the message content. We detail the structure of JMS messages, discuss key attributes, and consider when a given message type should be used. We also answer questions regarding how the JMS client uses the JMS message interface. For instance: How do I select a specific message of interest or populate it with content?

Message Definition

Message definition is probably the most crucial and potentially the most challenging of all activities associated with enterprise messaging. It defines the process of determining message content (what makes up a message) and structure (how message content is organized), without which applications do not have the basis to communicate. The varied nature and number of applications that interact using the enterprise messaging infrastructure implies that the number and form of message definitions can vary widely. This typically suggests a requirement for transformation services that transform messages between different formats. Such services are often provided by messaging providers as part of the delivery service.

In approaching the problem of message definition proliferation, some encourage the use of a common message model or canonical form that all messages exchanged are based on. Of course, the main challenge is defining a suitable canonical form that addresses the needs of all the enterprise applications. Irrespective of the approach to message definition adopted, the prob-

lem still remains to define for any given message the message content and its physical format or layout. Together, the message content and physical format define the data block that will be contained within the body of the JMS message.

The message content problem is typically bounded by the actual functions of the applications that will be producing or consuming the message. For instance, for an application that is responsible for submitting an order, it's safe to assume that the message content representing the order will contain data consistent with how an order is defined in that enterprise. The content should address the needs of the application that will process the order as well as take into account the data the submitting application has available. Expected content could comprise an order identifier, customer details, and a list of ordered items.

Defining message content can thus be viewed as an exercise in understanding and interpreting the business context. In addition, the potential interactions between the communicating applications must be considered. For example, when the order is submitted, how is the status of the order communicated? We could adopt the approach that our order message contains a status field, which is populated by the order fulfillment application. In this case we use a single message type for submitting an order as well as for receiving or querying its status. Alternately, we could design an order status message that is returned when the order is submitted and also used when the status of the order is requested. Note that in both cases we would also have to define the dependencies among the data. For instance, in the case where I use the same message type for different purposes, what fields are mandatory or optional based on my usage?

Sometimes the message content is predefined by the application that ultimately produces or consumes the message. This is often the case when enabling existing off-the-shelf or custom applications to communicate via messaging. As an example, the message content of a message designed to move data to or from a database table could be expected to bear a close relationship with the columns and rows defined for the database table.

The physical format of a message refers to the actual representation of the message content. It specifies how the data is structured and laid out and defines the relationship between discrete items of data, usually called fields or elements. The physical format enables the sending or receiving application to make sense of the data contained within the message.

As an illustration of the problem, consider the data block:

```
JOHN DOE 33
```

Is this a single data field? Or is it three fields, each separated by a space? What is the nature of the data? Is it a string? Is 33 an integer? The ways in which this data block could be interpreted are endless, and the physical format enables the data to be interpreted as intended.

To further our discussion of physical formats, we examine three physical formats commonly used to define messages: XML, tagged/delimited, and record-oriented.

XML

XML (Extensible Markup Language) is probably the most commonly used physical format in Java applications. It is a standards-based markup language (http://www.w3.org/XML/) that

allows data to be described independent of the language the application is written in or the operating system on which the application runs. The inherent portability of XML data is a major factor in its popularity in the Java world, which is understandable given that Java itself is aimed at developing portable applications that can run on any operating system. XML is also widely used by non-Java applications and is definitely experiencing a boom in its usage. Indeed, it may not be inappropriate to suggest that XML could be considered the physical format of choice, or at the very least the first port of call, for representing data in today's business applications.

XML is a markup language: it uses tags to identify pieces of data. XML, like its sibling HTML, owes its existence to Standard Generalized Markup Language (SGML), which was defined in 1986 by the International Standards Organization in the standard ISO 8879. SGML had its roots in document representation and was designed to provide a basis for describing the various elements associated with a document, including text, formatting, linking, and embedding. It introduced the concept of providing start and end tags to identify a piece of textual data. Hence, a first name of John would be described `<firstName>John</firstName>`, with the addition of the forward slash (`/`) identifying the end tag. The use of start and end tags meant that data could be nested and described in a hierarchical form. For instance, the attributes of a given individual could be described as:

```
<person>
     <firstName>John</firstName>
     <lastName>Doe</lastName>
     <age>33</age>
</person>
```

In essence, SGML provided a way to identify, name, and describe the relationship between pieces of data so they could be managed and manipulated. That said, the SGML standard was by no means simple, as it provided a generalized framework and was consequently extensive in scope and detail. A key characteristic of SGML is that it is a metalanguage, which means that it defines how any given markup language can be specified. HTML was first introduced in 1990 as a markup language for describing data that is rendered in browsers as Web pages. HTML defined a strict set of tags and associated syntax to be used and provided the impetus for the explosion of the Internet, as any browser that could process HTML could display the resulting data. XML, introduced in 1996, was aimed at providing a simpler and lightweight generalized markup language for use on the Web. It differed from HTML in one key aspect: it did not define a tag set. This meant that XML was highly extensible and flexible, as tags could be defined to suit any given purpose. In addition, as long as the XML document was well formed (e.g., matching start and end tags), the data could be processed by any application that could handle XML. Of course with such flexibility there was the question of whether the tags would be understood by the processing application, and a number of specifications, which we will examine later, such as DTD and Schemas, evolved to enable the meaning of an XML document to be shared among applications.

XML describes data in a textual form that is typically human readable, it is used in a variety of applications ranging from presentation of information and storage of configuration options, to communication of data—a use we are obviously interested in. Being a meta-language in its own right, XML has formed the basis of other language specifications that define an explicit tag set and associated syntax for their use. Examples include SOAP, a standards-based format for describing messages sent using Web services, and WSDL, a format for describing Web services.

Anatomy of an XML Document

Figure 3–1 shows an XML document (a block of data rendered in XML) with key attributes of the document highlighted.

An XML document typically commences with an optional XML declaration that in the least identifies the version of XML in the document and additionally the character encoding. If the XML declaration is omitted, the version is assumed to be 1.0, and heuristics are applied by the processing application to guess the encoding. The XML declaration is known as a processing instruction (PI) and is characterized by the syntax `<?PITarget instructions?>`. `PITarget` represents a keyword meaningful to the processing XML parser or application. In our example we specify a target `"XML"`, which refers to the XML standard set of PIs that a

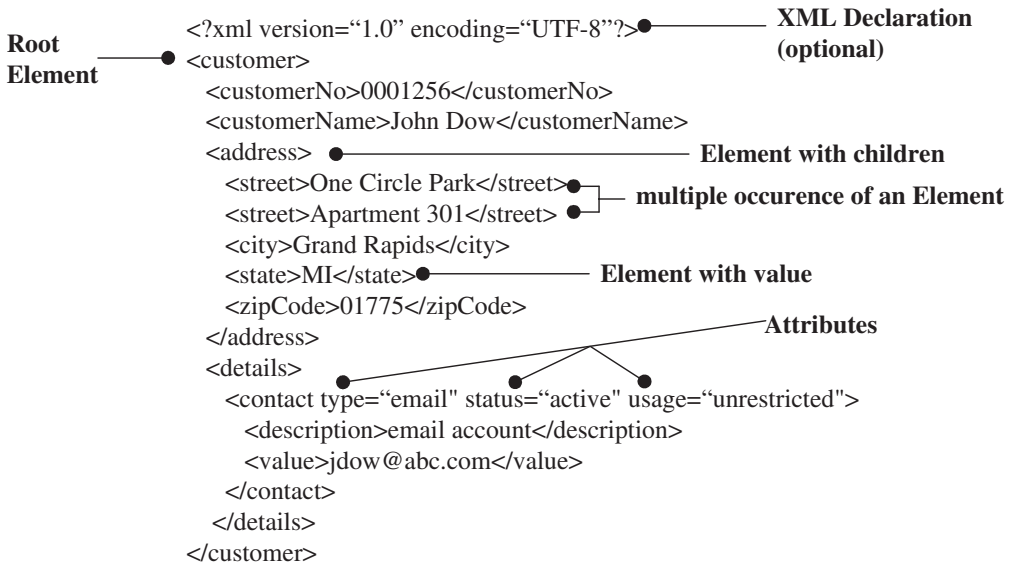

Figure 3-1 Anatomy of an XML Document

parser should recognize; the specific instructions identify version and character encoding. Application-specific PIs may be defined and included in the document.

The XML data is bound by a root tag `<customer> … </customer>`. The root tag encapsulates all other tags associated with the data block and marks the start and end of the XML data. As would be expected with most hierarchical structures, there can be only one root tag. An element is generally defined by a start tag, end tag, and some value. The element itself may contain nested elements and may indeed be empty; that is, they may have no value or nested element assigned. In the case of an empty element, a special shorthand may be used specifying a start tag ending in a forward slash, for example, `<value/>`. Elements may optionally contain attributes, which are name-value pairs that provide an alternative way of structuring data. For example, element `contact` has associated with it three attributes: `type`, `status`, and `usage`.

The question that always arises is: When do I represent data as an element or as an attribute, given that either approach can be easily adopted? For instance,

```
<contact type="email" status="active" usage="unrestricted">
….
</contact>
```

can just as easily be represented by

```
<contact>
<type>email</type>
<status>active</status>
<usage>unrestricted</usage>
…
</contact>
```

The simple answer is that it depends. It basically comes down to what constitutes a logical arrangement, what makes contextual sense, personal preference, and style.

In reviewing the XML example (Figure 3–1), you may have noted some simple syntactical rules. Element names must begin with a letter or underscore and may not contain embedded spaces. All start tags must have a corresponding end tag, and the order of nesting must be maintained. An XML document that conforms to these rules is said to be well formed; however, that a document is well formed does not mean that it is correct or valid from the perspective of the application that is processing it. As an example, assume I wish to send an XML-based message to a shipping application. The shipping application requires that a postal code (ZIP Code) be specified and the sending application sends the following:

```
<address>
      <street>One Circle Park</street>
      <street>Apartment 301</street>
      <city>Grand Rapids</city>
      <state>MI</state>
      <zipCode/>
   </address>
```

The XML document in itself is well formed, obeying all the syntactical rules; however, from the perspective of the shipping application, the XML document is not valid because it does not contain a required value: `zipCode` is empty. When viewed from the perspective of validity, you may have already begun to ask other questions, such as how the shipping application knows that the element `zipCode` is contained in the element `address`, or how the sending application can be notified that in preparing the XML document, `zipCode` is a mandatory element that must be populated. In other words, is there a blueprint or metadata that describes the structures and relationships in the XML document? This need is addressed by DTDs and XML Schema.

DTDs and Schemas

DTDs (Document Type Definitions) and XML Schemas define what an XML document should look like. They define the ordering of elements, mandatory and optional elements, allowable values, and a host of other characteristics that make it possible for an application to successfully interpret an XML document. While they are not required to successfully parse or construct an XML document, they define a reusable template for the XML document that can be shared among applications, and they provide a definition against which the document can be validated.

Initially, a DTD was the only means of describing an XML document, but as usage evolved, a number of restrictions became apparent:

- DTDs are not expressed in XML; they are simple text documents and are consequently not as easy as XML to process or manipulate.
- DTDs do not allow the value of an element to be typed as anything other than a `String`. While it was accepted that an XML document represented all data in textual form and was in itself one large `String`, users wanted the capability to say that the value of a given element was not actually a `String`, but an integer, for example, and should be treated as such by the processing application.
- DTDs do not provide a mechanism to define the scope of a tag to protect against name collision. For example, does the element `type` mean the same thing when defined as part of the element `account` and the element `phoneNumber`? Are the allowed values in both cases the same? And how do we address the ambiguity?

XML Schemas were introduced to address these and other limitations, and are the preferred option over DTDs. XML Schemas are written in XML. They allow elements to be typed and are closely associated with the concept of XML namespaces, which provide a solution to name collision. The concepts associated with XML Schemas, particularly XML namespaces, are nontrivial, and a detailed review is outside the scope of this book. However, a number of excellent references are suggested in the resource list in Appendix D. For our purposes, it suffices to say that armed with a DTD or Schema, our application can validate that the message content is appropriately formatted and populated before sending or on receipt. It is useful to note that from a performance viewpoint, the overhead associated with validating an XML document can be sig-

nificant. It is thus not unusual for validation to be done as part of development and testing, and disabled when the applications move into production, as thorough testing should have confirmed the correctness of the exchanged XML.

Processing XML Documents

Generally, an XML-aware application is concerned with constructing XML documents from various data structures and/or parsing XML documents into data structures that can be accessed and manipulated. From the JMS client's perspective, the XML document will be inserted into or retrieved from the message body of a JMS message. Given the textual nature of the XML document, it is fair to assume that it would be contained in a JMS `TextMessage` or `BytesMessage`.

A number of APIs have been developed in Java that enable an application to process an XML document. In all cases the use of the API is associated with a supporting XML parser that implements support for the API. The XML parser is responsible for actually parsing the XML document and optionally validating it against a Schema or DTD. A good example of an XML parser is the open source Apache Xerces2 Java Parser *(http://xml.apache.org/xerces2-j/index.html)*, which supports the full range of APIs. It is particularly popular among developers and is repackaged by a number of vendors, including IBM. Currently, Java developers have a choice of four APIs to use for XML processing: SAX, DOM, JAXP, and JDOM.

SAX SAX (Simple API for XML; *http://www.saxproject.org/*) focuses on the parsing of XML documents only. It does not provide for the building of an XML document from scratch or the modification of an existing document. It offers an event-driven interface that enables the application to register content handlers that are called with parsing events as they occur. The parsing events are triggered as the document is parsed by the parser, and the content handler implements methods (callbacks) that are called based upon the nature of the event. Examples include `startDocument()`, `endDocument()`, `startElement()`, `endElement()`, and `characters()`. In all cases the callback methods are passed arguments, which are processed by the application. The `characters()` method receives the actual data associated with a given element.

Under SAX, the parser handles the document sequentially, triggering events as it progresses. Once an event has been triggered, it cannot be retrieved, and SAX does not store processed values. This offers a performance advantage, as large documents can be easily handled without constraining available memory. The downside is that SAX provides no means of accessing data randomly or at a specified location. It also provides no in-memory model of the document that can be accessed by the application.

DOM The DOM (Document Object Model) API is aimed at providing an in-memory model of the XML document defined as a tree. A document object is provided by the API, which contains the parse tree and methods that enable the tree to be traversed. The API also allows a document object and thus a tree to be built from scratch, providing methods to create elements,

attributes, and other XML constructs. Unfortunately, the API does not provide a means to easily render a built tree in XML, and custom string-generation methods are required if DOM is to be used this way. Such utilities may be provided by the parser vendor; for example, Xerces provides an `XMLSerializer` class for this purpose.

The greatest concern when using DOM is the resource constraints that can occur when parsing large documents, as in all cases the entire document must be read into memory before it can be accessed. It is instructive to note that DOM actually uses SAX to parse the document as it builds the tree.

DOM was not designed specifically for Java, but is rather a general specification (see *http://www.w3.org/DOM/*) that can be implemented in any programming language. This has resulted in what some developer's have described as an API that is foreign to Java programmers. While this is of course a matter of opinion, the API does offer certain idiosyncrasies such as treating everything, even data, as a node on the tree (one would expect data to be associated with a node and not be a node itself), which can make the API feel unwieldy.

JAXP JAXP (Java API for XML Parsing) was introduced by Sun to attempt to provide some standardized abstraction to the use of SAX and DOM with different parsers. One of the drawbacks of SAX and DOM is that they both require that the XML parser is explicitly specified, which of course means that code would need to be modified if a different parser is used. While one could work around this limitation using property files, JAXP (http://java.sun.com/xml/jaxp/) provides a standard pluggability layer that allows applications to be written independent of the XML parser used. It allows the XML parser to be specified using Java system properties, and provides factory classes that enable a SAX or DOM parser to be invoked. It does not change the SAX or DOM APIs, but simply adds some convenience constructs that abstract the vendor specific parser.

JDOM The JDOM API is an open source effort *(http://www.jdom.org/)* that addresses the limitations of SAX and DOM. Despite the name similarity, JDOM is not based on DOM, though it adopts a document object model. It was designed and optimized specifically for Java and harnesses the power of the Java language to provide a simple API with credible performance characteristics. It is not dependent on an XML parser implementing support for JDOM, as it uses SAX (or DOM) to invoke the parser; thus it can be used with existing parsers. As it is based on a document object model, it allows the parse tree to be constructed from scratch, and unlike DOM, it provides utility classes that can render the tree in a number of forms, including XML.

While not intended to be a full-fledged tutorial on using JDOM, to round out our discussion on XML, the code snippet below illustrates the use of JDOM to construct and insert a simple XML document into a JMS message. It also shows how to parse an XML document retrieved from a JMS message.

```
//JDOM imports
import org.jdom.*; //XML Processing API
import org.jdom.input.SAXBuilder; //Parser
```

```
import org.jdom.output.XMLOutputter;//XML generator

/*Sender*/
//build XML document from data
String firstName = "JOHN";
String lastName = "DOE";
int age = 33;

//create document object
Document doc = new Document(new Element("person"));
//retrieve root element and build tree
Element root = doc.getRootElement();
root.addContent(new Element("firstName").setText(firstName));
root.addContent(new Element("lastName").setText(lastName));
root.addContent(new
Element("age").setText(Integer.toString(age)));
//output tree as XML stream ready for transmission
XMLOutputter fmt = new XMLOutputter();
//XMLOutputter output method works with java.io.OutputStream
//though an outputString method is provided, its use is not
recommended by jdom
ByteArrayOutputStream out = new ByteArrayOutputStream();
fmt.output(doc, out);

//create TextMessage
TextMessage tOutMsg = session.createTextMessage();
tOutMsg.setText(new String(out.toByteArray()));

/*Receiver*/
//Unpack TextMessage
String message = tInMsg.getText();

//parse message content to generate document
ByteArrayInputStream in = new
ByteArrayInputStream(message.getBytes());
//create parser using default (Xerces) and no validation
SAXBuilder parser = new SAXBuilder(false);
Document doc = parser.build(in);
Element root = doc.getRootElement();
String firstName = root.getChildText("firstName");
String lastName = root.getChildText("lastName");
int age = Integer.parseInt(root.getChildText("age"));
```

Note that for the construction and parsing of the message, JDOM favors `java.io.InputStream`, and I used the `ByteArrayInputStream` and `ByteArrayOutputStream`. My reason for not simply using a `BytesMessage` (which would be a natural fit) is specifically related to the fact that with the `BytesMessage`, the receiver does not

know the size of the incoming message, which hampers its ability to extract the contents (a problem fixed in JMS 1.1). This is discussed in detail in the section "Using the JMS Message Interface," later in this chapter.

Tagged/Delimited

A tagged/delimited physical format uses tags, delimiters, or both to define the structure of a block of data, which is usually rendered as text. If we revisit our sample message data, `JOHN DOE 33`, we could define this as three fields delimited by a space. Alternately, `JOHN;DOE;33` would be three fields delimited by a semicolon. Tagged/delimited physical formats have been in use for years and are found in many industrial-strength applications. For instance, the SWIFT message format, which is used to move money electronically between banks, is tagged/delimited. Automated commerce transactions between businesses often rely upon the exchange of messages that conform to the Electronic Data Interchange (EDI) format, which is also traditionally a tagged/delimited format. Indeed, in the strictest sense XML can be thought of as a formalized tagged/delimited format.

Using a tagged/delimited format simply comes down to deciding if your fields need to be tagged and how the fields are delimited. A number of approaches can be adopted to define the structure of the data:

Variable-Length Delimited

With a variable-length delimited approach, the data fields are separated from each other by a delimiter. The delimiter is a constant that marks the end of a field. Punctuation constructs such as commas (,), full stops (.), semicolons (;), and colons (:) are commonly used. This approach proves particularly useful when the length of a data field varies from message to message (hence the term variable-length). For example, the length of a person's name varies widely. Defining a fixed length for the data field that will contain a name requires you to define an allowable maximum. This runs the risk of someday proving inadequate or alternately proves inefficient because the full length is never used and thus a great portion of the message is essentially redundant space.

It is not uncommon to have data that contains a combination of fixed-length and variable-length fields. You can choose to simply delimit all fields using a delimiter, or you can choose to delimit fixed fields based on their length and variable fields with a delimiter. For example, in

```
1050JOHN;DOE;33
```

`1050` is a value with a fixed length of four characters. The other fields are variable and delimited with a semicolon.

Tagged Delimited

As the name suggests, tagged delimited uses a tag to identify data fields. The tag generally precedes the data field and is itself separated from the data field either by a known tag length or by a specified tag data separator, which is essentially a delimiter that specifies the end of the tag. The data field is separated from the next tag by a specified delimiter. For example, in

```
DATA:0001;DATA:0002;DATA:0003
```

DATA is the tag for each field, a colon is the tag data separator, and a semicolon delimits the data field from the next tag. A variation on this is tagged fixed length, in which the data fields are delimited by their known length.

It is certainly possible for techniques to be combined when defining data formats, and varied and complicated variations can occur. For example, if a field is delimited by its length, the data structure may be designed such that the length of the field is carried as part of the message, as shown:

```
X:4;0001Y:6;000123
```

where X is a tag separated from the following data field by a colon. The data field containing the value 4 is delimited by a semicolon from a fixed data field whose length is the value contained in the preceding field: 4. The data structure then repeats the same pattern with a tag, Y, and a data field whose length is 6.

Processing Tagged/Delimited Formats The tagged/delimited physical format offers considerable flexibility in defining the way data is structured. Consider the sample SWIFT message shown:

```
{1:F01IBMADEF0AXXX0000000000}{2:I100IBMADEF0AXXXN}{3:{108:abc}
}{4:
:20:X
:32A:940930USD1,
:50:X
:52A:CHASUS33
:53A:CHASUS33
:54A:CHASUS33
:56A:CHASUS33
:57A:CHASUS33
:59:X
:70:X
:71A:OUR
:72:/A/
-}
```

The message is defined by blocks of data grouped together. Each block is identified by a tag (1, 2, …) which is separated from its data fields by a colon. Blocks are delimited by curly braces (}{). Notice that in the block tagged with a 4 (known as the SWIFT message body; the

others are headers), each data field is further tagged with a colon as separator, and the data is delimited with a carriage-return linefeed.

Clearly, the parsing or construction of such a message format is nontrivial; however, it illustrates an important point: if you decide to define a tagged/delimited format, you must typically design code to successfully parse and construct messages in that format. As an example, the following code snippet illustrates how our sample variable-length delimited message `JOHN;DOE;33` might be parsed:

```
//parse received message
String message = tInMsg.getText();
//initialize variables
String firstName = "";
String lastName = "";
int age = 0;

//use java.util.StringTokenizer to parse string
StringTokenizer st = new StringTokenizer(message, ";");
while (st.hasMoreTokens()) {
     firstName = st.nextToken();
     lastName = st.nextToken();
     age = Integer.parseInt(st.nextToken());
}
```

It is useful to note that in the case of well-known, complicated formats such as SWIFT and EDI, vendor-supplied tools are generally available to assist in the processing of such data.

Why, you may ask, would I want to create a tagged/delimited format when I can use XML with all its associated tools? Performance considerations and optimizing message size are influencing factors. XML by design is a verbose format, and as previously discussed, processing XML documents can attract significant overhead. In contrast, representing the data using a tagged/delimited format such as variable-length delimited results in much less data that needs to be transmitted and, with careful choices, a structure that is not that hard to parse or construct. However, this is usually at the expense of readability and maintainability. Thus, in choosing a physical format, thought should be given to the use of the message and the impact the choice of physical format will have on development effort, performance, maintainability, ease of change, and ease of data sharing (remember that the physical format definition will need to be shared with at least one other application).

Record-Oriented

The record-oriented physical format is a fixed length–based format; that is, data is described based on the length of associated typed fields. For example, `JOHNDOE33` is described as two strings of length 4 and 3 and an integer. Record-oriented formats were first introduced by mainframe-based applications as a means of formatting data that was to be exchanged between programs. Such applications were often written in COBOL, and COBOL copybooks

defined the structure of data in terms of the number of bytes representing each field. For applications written in C, the *struct* provided a useful construct for representing record-oriented data. In Java, record-oriented formats are often rendered in byte arrays, and the JMS `BytesMessage` object provides useful methods for rendering record-oriented physical formats, as we see later in this chapter.

While a record-oriented format can form the most concise description of a data block, this is done at the expense of readability and ease of sharing. For example, it is not as straightforward to share the definition of a record-oriented format between two applications that are written in different languages as it is to share an XML Schema. As discussed earlier, if fields may vary in length, then a maximum allowable length must be defined, with the attendant risks. Consideration also must be given to padding characters (often null or whitespace), which must be inserted when the actual data is less than the maximum length. The question of justification—Is data inserted from the beginning or end of the field so that blank space is either after or before the data?—also needs to be considered.

As with XML and tagged/delimited formats, record-oriented formats can be used to render quite complex data structures, and I would not want to convey the impression that they are obsolete, as they are widely used by today's enterprise applications. However, it is fair to say that new Java applications being developed today will most likely use XML rather than a record-oriented format for reasons discussed earlier. In the instance where the JMS application is sending or receiving a message from a non-JMS application that uses a record-oriented format, we may find ourselves handling record-oriented formats directly. However, recall that a messaging provider may offer value-added transformation services, which transform messages into physical formats appropriate for the recipient. If such services are available and are used, then the physical formats adopted by different applications do not need to be constrained. A given application simply speaks in a format that best suits its purposes.

JMS Message Structure

As discussed in Chapter 2, "Java Message Service," the JMS message is composed of a header, properties, and body, and it is typed based on the message data contained within the message body. The JMS message class hierarchy is organized such that the `Message` interface, which is the root interface for all the JMS message types, defines the header fields, property facility, and associated accessor methods. The `Message` interface is extended by the body-specific message interfaces—`BytesMessage`, `TextMessage`, `StreamMessage`, `MapMessage`, and `ObjectMessage`—which define the accessor methods associated with their specific body type.

It should be noted that in the interest of conciseness and clarity, when discussing attribute types I use the common interface classes (see Table 2–1), such as `Destination`. Depending on the messaging domain being used, the corresponding domain-specific interface, such as `Queue` or `Topic`, could be substituted.

Message Header

The message header is comprised of the following fields, which are present in every JMS message. Setter and getter methods are provided for all fields, but as detailed, setting a value often has no effect if the message is being sent and the field is populated by the provider on completion of the send.

`JMSDestination`: `JMSDestination` holds a `Destination` object that represents the destination to which the message is being sent—for example, `Queue` or `Topic`. Its value is set by the provider after completion of the send and matches the destination associated with the `MessageProducer`. A getter method is defined to allow the receiving application to access the value of `JMSDestination`. However, although a setter method is provided, the value of `JMSDestination` is ignored during a send.

`JMSDeliveryMode`: `JMSDeliveryMode` is used to define the persistence of a message. A nonpersistent message is not logged to a persistent message store by the provider, implying that the message cannot be recovered in the event of a provider process failure. Persistent messages, on the other hand, are logged and are thus recoverable by the provider when it restarts. `JMSDeliveryMode` holds an integer value defined by the interface `DeliveryMode`. `DeliveryMode` contains two static integer values: `NONPERSISTENT` and `PERSISTENT`. The value of `JMSDeliveryMode` is set by the provider after completion of the send. It is based on the value specified by the sending method, which can be either `DeliveryMode.NONPERSISTENT` or `DeliveryMode.PERSISTENT`. The provider defaults to persistent delivery if no value is specified. As with `JMSDestination`, any value set by the setter method is ignored on send. A getter method provides access to the populated value.

`JMSExpiration`: `JMSExpiration` stores the expiration time of the message. This is calculated as the current time plus a specified timeToLive. The value of timeToLive is specified on the sending method in milliseconds, and the provider populates `JMSExpiration` (type `Long`) with the computed value in milliseconds on completion of the send. If a value for timeToLive is not specified or is specified as zero, `JMSExpiration` is set to zero, signifying that the message does not expire. Any value set for `JMSExpiration` is ignored on the send. Once a message has expired, the JMS client is unable to retrieve the message. Typically, the message is discarded by the provider at this point. `JMSExpiration` provides a convenient way to clean up messages that have a well-defined relevance period, such as a reply to a request that has arrived late.

JMSPriority: JMS defines transmission priority levels 0 to 9, with 0 as the lowest priority and 9 as the highest. It additionally specifies that priorities 0 to 4 should be considered gradations of normal delivery and priorities 5 to 9 as gradations of expedited delivery. The provider populates `JMSPriority` (type `int`) on completion of the send based on the value specified by the sending method.

JMSTimeStamp: The `JMSTimeStamp` field is populated by the provider during the send; the value set is in the format of a normal Java millisecond time value.

JMSMessageID: `JMSMessageID` uniquely identifies a message. It is defined as a `String` and ignored if set when the message is sent. It is populated with a provider-assigned value on completion of the send. The provider-assigned identifier is prefixed with `ID:`.

JMSCorrelationID: The `JMSCorrelationID` field is typically used to link two messages together. For example, in order to link a given reply to a request, the JMS client could specify that the `JMSCorrelationID` of the incoming reply should match the `JMSMessageID` of the sent request. This requires the servicing client to set the `JMSCorrelationID` of the reply to the `JMSMessageID` of the request before sending the reply. Consequently, if used, this field is set by the JMS client, as shown:

```
//set correlationID
replyMsg.setJMSCorrelationID(requestMsg.getJMSMessageID());
```

JMS additionally allows for the case where the JMS client needs to use a specific value for linking messages; in this case an alternative string value can be set, but it should not be prefixed with `ID:`. If the JMS client is servicing a request from a non-JMS client and needs to assign a specific value for use with such clients, JMS defines a byte array optional type for `JMSCorrelationID`. However, providers are not required to implement support for this optional type.

JMSReplyTo: `JMSReplyTo` is set by the client and takes as value a `Destination` object that specifies where replies to this message are to be sent. This implies that this field is set only if a reply is required. By default it is set to null. `JMSReplyTo` can be set as shown:

```
//set reply to destination
Queue replyQ = (Queue)ctx.lookup(jmsReplyQ);
requestMsg.setJMSReplyTo(replyQ);
```

`JMSType`: `JMSType` contains a message type identifier that is set by the client. Its purpose is to enable clients of providers who provide a message definition repository to associate with the contained message body a reference to its physical format definition in the provider's message repository.

`JMSRedelivered`: The `JMSRedelivered` field takes a Boolean value set by the provider to signify that the message has been previously delivered to the client. It is only relevant when consuming a message and serves to alert the client (if checked) that it has previously attempted processing of this message. Recall from our discussion of message consumers and "poison" messages in Chapter 1, "Enterprise Messaging," that `JMSRedelivered` can help address the issue of infinite repeated processing, which could occur if the message is being processed within a transaction and being redelivered because processing failed.

Overriding Message Header Fields

JMS allows providers to provide a means for JMS administrators to override the client's settings for `JMSDeliveryMode`, `JMSExpiration`, and `JMSPriority`. This could be to enforce company policy or adopted best practice, or to change the behavior of the client without having to change code. JMS does not define specifically how this facility should be implemented, but IBM JMS providers utilize the administered objects, specifically the `Destination` object, to provide this feature. As discussed in Chapter 2, administered objects are stored outside the application in a JNDI namespace and contain vendor-specific settings. The `Destination` objects defined for IBM JMS providers allow values for `JMSDeliveryMode`, `JMSExpiration`, and `JMSPriority` to be set. By default they are set to "application specifies," but if set to an explicit value, they override any settings specified by the client on the sending method. We review the IBM JMS–administered objects in detail in Chapter 6, "IBM JMS–Administered Objects."

Properties

Properties are used to add optional fields to the header. A property can be specified by the application, be one of the standard properties defined by JMS, or be provider-specific. When sending a message, properties can be set using associated setter methods. In the received message, properties are read-only, and any attempt to set the properties on a received message results in a `MessageNotWriteableException`. If it is desired to modify the properties of a received message—for instance, the application is performing a brokering function and needs to modify the attributes of the message before forwarding it to the next destination—then the `clearProperties()` method is called on the message object.

Application-Specific Properties

Consider an audit application that tracks all orders being sent but wants to log an entry only if the order value is over a certain amount. An approach to solving this problem is to have the application process every order message passed to it and check the contents of the message to ascertain if the order value is above the threshold. A more efficient approach is to have the client application tell the provider that of all the messages sent to it, only those with an order value greater than some specified amount should be delivered. Application-specific properties enable this approach by offering a basis for a provider to filter messages based on application-specific criteria.

Continuing with our example, the sending application defines and assigns a value to an application-specific property, which we call `orderValue`. The message is then sent. The audit application defines a message selector (more on this in the next section) that details the selection criteria based on the application-specific property `orderValue`. The provider subsequently passes the sent message to the audit application only if the selection criteria are satisfied.

Why go through all this? Why not simply specify the selection criteria based on the "total cost" field in the message? The answer is quite simply that message selectors cannot operate on the message body. If message selectors did operate on the message body, then providers would have to have a means to successfully parse each message (a nontrivial task; see "Message Definition"), and we would be faced with the same performance bottleneck that having the application check each message would attract. In addition, this approach assumes that the data we want to filter on is contained in the message, which is not necessarily the case.

Handling the filter property as an optional header field is obviously much more efficient, as the provider understands the structure of the header, and given the typically small size of the header relative to the body, the property can be accessed much more quickly. Application-specific properties are typed and can be `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, and `String`. The properties are set by the sending application on the outbound message. For example,

```
//set property on outgoing message
outMessage.setIntProperty("orderValue", 3000);
```

Setter methods exist for each type, and they accept as arguments the property name and the associated value. A `setObjectProperty()` method is provided if the objectified version of the primitive types is to be used. As would be expected, corresponding getter methods are defined. Application-specific properties are used not only for selection criteria, but for a wealth of reasons, such as a means of specifying processing flags that can't be carried in the message. However, it is for use in message selection that they are most often considered. A list of application-specific properties defined for a message can be retrieved by calling `getPropertyNames()` on the message, which returns an `Enumeration` object.

Standard Properties

Standard properties are additional header fields defined by JMS, which further define the message. Apart from two noted exceptions, `JMSXGroupID` and `JMSXGroupSeq`, support for all other standard properties is optional. To determine which are supported by a given JMS provider, you can obviously read provider-specific documentation or use `ConnectionMetaData.getJMSXPropertyNames()` as shown:

```
// retrieve factory
ConnectionFactory factory = /*JNDI LOOKUP*/
// create connection
Connection connection = factory.createConnection();
//retrieve information describing the connection
ConnectionMetaData connInfo = connection.getMetaData();
//getJMSXPropertyNames
Enumeration propNames = connInfo.getJMSXPropertyNames();
```

`ConnectionMetaData` is a useful JMS interface that provides information that describes a JMS provider's implementation. It offers access to details such as JMS version, provider name, and other information. The IBM JMS providers support the following standard properties.

`JMSXUserID`: `JMSXUserID` contains the identity of the user sending the message. It is set by the provider on send and usually resolves to the user ID under which the client application is running. It is defined as a `String`.

`JMSXAppID`: `JMSXAppID` is the identifier associated with the application sending the message. It is similarly set by the provider on send. It is defined as a `String`.

`JMSXDeliveryCount`: `JMSXDeliveryCount` defines the number of times that a message has been delivered. It complements `JMSRedelivered`, which simply signifies that a message has been previously delivered, by providing a count of the number of times delivery as been attempted. It is set by the provider when the message is received.

`JMSXGroupID` and `JMSXGroupSeq`: `JMSXGroupID` and `JMSXGroupSeq` together uniquely define a message as belonging to a particular group of messages and being at a certain position (sequence) in the group. They are set by the client before sending and are defined using the `set<type>Property` methods as shown (they are defined as a `String` and `int` respectively):

```
//set group properties on outgoing message
outMessage.setStringProperty("JMSXGroupID", "10000001");
outMessage.setIntProperty("JMSXGroupSeq", 1);
```

Provider-Specific Properties

As the name suggests, JMS offers a means for providers to include specific fields in the header. The stated purpose is to support communication with non-JMS clients that might require certain properties set. Thus, they should never be used when knowingly communicating between two JMS clients. As would be expected, IBM JMS providers do specify a number of provider-specific properties, given that the messaging provider can be accessed using a variety of APIs and languages (see Chapter 5, "IBM JMS Providers"). Provider-specific properties are prefixed by `JMS_<vendor_name>`, in our case `JMS_IBM`; however, the properties are not detailed here because they do not add much to our JMS-specific discussion at this time. In Chapter 7, "JMS Implementation Scenarios," where we examine a usage scenario that involves knowingly communicating with a non-JMS client, we explore the use of provider-specific properties in a more appropriate context.

Message Selectors

A JMS message selector is used by a JMS client to specify which messages it is interested in. It is based on values contained in the message header, which includes standard header fields as well as optional fields that have been added via application-, standard-, or provider-specific properties. As noted earlier, a message selector cannot refer to message body values. A message selector is a Boolean expression that, when it evaluates to true, results in the matched message being passed to the client. It is defined as a `String`, and its syntax is based on a subset of the SQL92 conditional syntax.

Creating a selector involves generating a `String` that conforms to the defined syntax. For instance, continuing with our previous example of the audit application that wants to receive messages only if the `orderValue` (specified as an application-specific property) is greater than a certain threshold, the message selector would be of the form

```
//Set selector
String selector = "orderValue > 2500";
```

Another common use of message selectors is to match a reply message to the original request based on the `JMSCorrelationID`, as shown (note that the syntax requires quotes around `Strings`):

```
//Set selector
String messageID = requestMsg.getJMSMessageID();
String selector = "JMSCorrelationID ='" + messageID + "'";
```

The syntax for message selectors enables pattern matching whereby the selector can match a variety of messages. For example, to match all messages that have a postal code that begins with SO—such as SO53 2NW and SO51 2JN—the selector would be of the form

```
//Set selector
String selector = "postalcode LIKE 'SO%'";
```

Given that a message selector is a Boolean expression, expressions can be combined using constructs such as AND or OR:

```
//Set selector
String selector = "stock = 'IBM' OR stock = 'Microsoft' AND
price = 100";
```

Once a message selector is defined, it is associated with the MessageConsumer that will check the specified destination for messages (we examine this in more detail in Chapter 4, "Using the JMS API"):

```
//get reply
QueueReceiver receiver = session.createQueueReceiver(replyQ,
selector);
TextMessage replyMsg = receiver.receive();
```

JMS enables fairly sophisticated patterns to be defined for message selectors, and the rules regarding selector syntax defined by JMS are extensive. They are reproduced in Appendix A for your convenience. In all cases it is important to remember that message selectors can be specified only on header values and not the message body. It is thus not uncommon to find application-specific properties being defined that duplicate certain fields in the message body so that the message can be selected based on what is essentially message content.

Message Body

As discussed in Chapter 2, JMS supports five types of message bodies, with each body type represented by a different message interface: BytesMessage, TextMessage, StreamMessage, MapMessage, and ObjectMessage. The choice of message type used is to a great extent, but not exclusively, governed by the nature of the data being sent. To recap,

- BytesMessage stores data as a sequence of bytes, supports the most basic representation of data, and thus can contain varied payloads.
- TextMessage stores data as a String, a common representation of enterprise data.
- StreamMessage stores data as a sequence of typed fields, for example, JohnDoe33, defined as a sequence of two strings (John, Doe), and an integer (33).
- MapMessage stores data as a set of key-value pairs. The key is defined as a string, and the value is typed—for example, Age:33, where Age is the key and 33 the integer value.
- ObjectMessage stores a serialized Java object.

BytesMessage and TextMessage are probably the most common type of message used. Both handle data in a form that facilitates the easy exchange of data across the enterprise between JMS applications as well as non-JMS applications. The common physical formats adopted—XML, tagged/delimited, and record-oriented—readily lend themselves to byte or string representations and thus can be easily transported using BytesMessage or TextMes-

sage. In particular, a record-oriented format rendered in a byte array is often the format of choice for exchanging data with legacy applications.

`ObjectMessage` is specialized in its use, as it stores a serialized Java object. Clearly, the recipient must be able to deserialize the received Java object, and thus the use of this message type is generally associated with JMS-to-JMS communication. The representation of data as a Java object can be efficient from the JMS client's point of view, as no explicit parsing or construction of the message body is required. The exchange of the object's class definition between JMS clients is also a trivial exercise, as this is part of everyday Java programming practice. However, when you consider messaging within an enterprise, rarely are all the interacting applications written in Java. More importantly, even if the ultimate destination is a JMS client, the message may pass through other infrastructure applications, such as message trackers that are not Java-based. Even when the target application is Java-based, another consideration, particularly with off-the-shelf packages, is whether the application supports the use of class definitions as the basis for defining data that will be exchanged. In truth, you are more likely to find support for XML than for Java objects. Consequently, the use of the `ObjectMessage` places more restrictions on who can ultimately process the contained data (object) and may not provide as flexible a solution in comparison to one built on the exchange of a non-language-based format such as XML.

`StreamMessage` and `MapMessage` are rather unique in that while JMS defines the interface, it does not actually specify the associated physical construct—string, serialized object, or byte array—in which the typed sequence or key-value pairs are contained. The stated reason for this is that JMS provides these message types in support of JMS providers whose native implementation supports some form of self-defining format, the idea being that the provider renders the `StreamMessage` or `MapMessage` in its native format, facilitating the exchange of data with native clients—that is, non-JMS clients that use the provider's native (proprietary) API. Given that the IBM JMS providers do not have a native self-defining format (data is generally treated as being opaque), the value of `StreamMessage` and `MapMessage` for this purpose is questionable.

With the IBM JMS providers, the contents of `StreamMessage` and `MapMessage` are rendered in XML, a standards-based self-describing format.

A `StreamMessage` is rendered in the following format:

```
<stream>
<elt dt='datatype '>value</elt>
<elt dt='datatype '>value</elt>
.....
</stream>
```

Every field is sent using the same tag name, `elt`, where `elt` contains an XML attribute, `dt`, that specifies the data type of the field in the sequence and has as value the actual contents of the field. The default type is `String`, so `dt='string '` is omitted for string elements.

A MapMessage takes the form:

```
<map>
<elementName1 dt='datatype '>value</elementName1>
<elementName2 dt='datatype '>value</elementName2>
.....
</map>
```

In this case, elementName1 ..... N maps to the key in the key-value pair.

Given that StreamMessage and MapMessage are rendered as XML by the IBM JMS providers, they do enable the exchange of data in a flexible form. In the case where the user does not want to develop XML data formats, they provide one out of the box that can be readily used. From the perspective of the JMS client, they do insulate the client from the fact that the data is XML, providing a simple interface to manipulate data, and unlike for ObjectMessage, the data can be handled by non-JMS and non-Java clients. However, because the implementation of StreamMessage and MapMessage is provider-specific, they should be used carefully. If it is desired to exchange data in XML, then defining an XML structure and using a BytesMessage or TextMessage provides a more generic, flexible, and portable approach.

As with properties, the body of a received message is read-only and cannot be directly modified by the receiving application. If the application wishes to populate the body of a received message with data, then it calls the clearBody() method on the message.

Using the JMS Message Interface

A message is created using the Session interface (see Chapter 2), and the Session interface defines create methods for all the message types, including the root Message interface (Table 3–1).

Table 3–1 Accessor Methods for Session Interface

```
createBytesMessage()
createMapMessage()
createMessage()
createObjectMessage()
createObjectMessage(java.io.seriablizable object) //create with content
createStreamMessage()
createTextMessage()
createTextMessage(String value) // create with content
```

As shown, createObjectMessage and createTextMessage are overloaded so that they can be instantiated with data at the time of creation. Creation of a message takes the form

```
//create message
BytesMessage bMsg = session.createBytesMessage();
```

If a message is being received, then the `MessageConsumer` creates the message and returns it to an object variable:

```
//receive message
TextMessage replyMsg = (TextMessage)receiver.receive();
```

This assumes that the type of message being received is known. In our example, if the message retrieved was actually a `BytesMessage`, then an exception would be thrown. Another approach is to use the root interface to receive the object variable, and then test the object and cast to the allowed type:

```
//receive message
Message msg = receiver.receive();
if(msg instanceof TextMessage){
     TextMessage tMsg = (TextMessage)msg;
     ......
}
```

The root `Message` interface defines the accessor methods for the header fields and properties; the typed `Message` interfaces extend this to include accessor methods for manipulating the data they contain in their message body. We examine the methods and sample usage for each of the typed interfaces in turn.

BytesMessage

Table 3–2 details the accessor methods associated with `BytesMessage`.

Table 3–2 Accessor Methods for BytesMessage Interface

`readBoolean()`	`writeBoolean(boolean value)`
`readByte()`	`writeByte(byte value)`
`readBytes(byte[] value)`	`writeBytes(byte[] value)`
`readBytes(byte[] value, int length)`	`writeBytes(byte[] value, int length)`
`readChar()`	`writeChar(char value)`
`readDouble()`	`writeDouble(double value)`
`readFloat()`	`writeFloat(float value)`
`readInt()`	`writeInt(int value)`
`readLong()`	`writeLong(long value)`
`readUnsignedByte()`	`writeObject(Object value) //works only for object primitive types`
`readUnsignedShort()`	`writeShort(short value)`
`readUTF()`	`writeUTF(String value)`

Use of these methods follow a simple rule: If you build the message body by using write methods in a certain order, then you unpack the message body by using the corresponding read

methods in the same order. Let's use our familiar John Doe example laid out in a record-oriented format to illustrate:

```
/*Sender*/
//create and send BytesMessage
String firstName = "JOHN";
String lastName = "DOE";
int age = 33;

BytesMessage bOutMsg = session.createBytesMessage();
bOutMsg.writeBytes(firstName.getBytes());
bOutMsg.writeBytes(lastName.getBytes());
bOutMsg.writeInt(age);
sender.send(bOutMsg);

/*Receiver*/
//receive and unpack BytesMessage
BytesMessage bInMsg = (BytesMessage)receiver.receive();
byte[] data = new byte[4];
bInMsg.readBytes(data, 4);
String firstName = new String(data);
bInMsg.readBytes(data, 3);
String lastName = new String(data, 0, 3);
int age = bInMsg.readInt();
```

If we want to retrieve the entire contents of `BytesMessage` using `readBytes(byte [] value)`, we must determine the length of the message body so that we can appropriately initialize the byte array that will contain the read data. Unfortunately, JMS 1.0.2b does not provide any means for determining the length of the message body, and if the length of the incoming message is not known, we may be forced to adopt processing of the form

```
//receive and unpack BytesMessage
BytesMessage bInMsg = (BytesMessage)receiver.receive();
int msgLength = 0;
int BUF_SIZE = 50000; //some efficient maximum
byte[] data = new byte[BUF_SIZE];
while(true){
     int len = bInMsg.readBytes(data);
     if(len > 0){
     msgLength += len;
     }else{
     break;
     }
}
// now we know the message length
// so reset and read in one go.
byte[] message = new byte[msgLength];
//if msgLength <= BUF_SIZE, then we already
```

```
//have the contents
if (msgLength <= BUF_SIZE) {
          System.arraycopy(data, 0, message, 0, msgLength);
} else {
       bInMsg.reset();//reset cursor to beginning
       bInMsg.readBytes(message);
}
```

Notice that we introduce the utility `method reset()`, which repositions the stream of bytes to the beginning and allows us to read in the entire byte array. JMS 1.1 addresses this issue by specifying a `getBodyLength()` method in the `BytesMessage` interface, which allows us to greatly simplify the process of dynamically sizing our byte array as follows:

```
//receive and unpack BytesMessage
BytesMessage bInMsg = (BytesMessage)receiver.receive();
int msgLength = bInMsg.getBodyLength();
byte[] message = new byte[msgLength];
bInMsg.readBytes(message);
```

TextMessage

As shown in Table 3–3, `TextMessage` offers an interface that is quite simple to use.

Table 3–3 Accessor Methods for TextMessage Interface

`getText()`	`setText(String value)`

It provides a `setText` and a `getText` method, which enable the message body to be populated or content extracted:

```
/*Sender*/
//create and send TextMessage
String firstName = "JOHN";
String lastName = "DOE";
String age = "33";

TextMessage tOutMsg = session.createTextMessage();
String message = firstName + ";" + lastName + ";" + age;
tOutMsg.setText(message);
sender.send(tOutMsg);

/*Receiver*/
//receive and unpack TextMessage
TextMessage tInMsg = (TextMessage)receiver.receive();
String message = tInMsg.getText();
```

StreamMessage

`StreamMessage` represents a sequence of primitive types, as shown in Table 3–4.

Table 3–4 Accessor Methods for StreamMessage Interface

`readBoolean()`	`writeBoolean(boolean value)`
`readByte()`	`writeByte(byte value)`
`readBytes(byte[] value)`	`writeBytes(byte[] value)`
`readChar()`	`writeBytes(byte[] value, int offset, int length)`
`readDouble()`	`writeChar(char value)`
`readFloat()`	`writeDouble(double value)`
`readInt()`	`writeFloat(float value)`
`readLong()`	`writeInt(int value)`
`readObject()`	`writeLong(long value)`
`readShort()`	`writeObject(Object value)`
`readString()`	`//works only for object primitive types`
	`writeShort(short value)`
	`writeString(String value)`

It provides methods that allow the various primitives to be written or read:

```
/*Sender*/
//create and send StreamMessage
String firstName = "JOHN";
String lastName = "DOE";
int age = 33;

StreamMessage sOutMsg = session.createStreamMessage();
sOutMsg.writeString(firstName);
sOutMsg.writeString(lastName);
sOutMsg.writeInt(age);
sender.send(sOutMsg);

/*Receiver*/
//receive and unpack StreamMessage
StreamMessage sInMsg = (StreamMessage)receiver.receive();
String firstName = sInMsg.readString();
String lastName = sInMsg.readString();
int age = sInMsg.readInt();
```

As with `BytesMessage`, `StreamMessage` also provides a reset method with which the stream can be repositioned to its beginning.

MapMessage

The `MapMessage` interface (Table 3–5) introduces a name variable that represents the key associated with the typed field. Because data is accessed based on the name (key), it does allow random access to data fields.

Table 3–5 Accessor Methods for MapMessage Interface

`getBoolean(String name)`	`setBoolean(String name, boolean value)`
`getByte(String name)`	`setByte(String name, byte value)`
`getBytes(String name)`	`setBytes(String name, byte[] value)`
`getChar(String name)`	`setBytes(String name, byte[] value, int offset, int length)`
`getDouble(String name)`	`setChar(String name, char value)`
`getFloat(String name)`	`setDouble(String name, double value)`
`getInt(String name)`	`setFloat(String name, float value)`
`getLong(String name)`	`setInt(String name, int value)`
`getObject(String name)`	`setLong(String name, long value)`
`getShort(String name)`	`setObject(String name, Object value)`
`getString(String name)`	`setShort(String name, short value)`
	`setString(String name, String value)`

To facilitate key management, the `MapMessage` interface offers two utility methods: `itemExists` and `getMapNames`. `itemExists(String Name)` takes as argument a key name and checks for its existence, returning a Boolean. `getMapNames()` returns an `Enumeration` object containing all the key names defined in the message. The use of `MapMessage` follows the now familiar pattern:

```
/*Sender*/
//create and send MapMessage
String firstName = "JOHN";
String lastName = "DOE";
int age = 33;

MapMessage mOutMsg = session.createMapMessage();
mOutMsg.setString("first", firstName);
mOutMsg.setString("last", lastName);
mOutMsg.setInt("age", age);
sender.send(mOutMsg);

/*Receiver*/
//receive and unpack MapMessage
MapMessage mInMsg = (MapMessage)receiver.receive();
String firstName = mInMsg.getString("first");
String lastName = mInMsg.getString("last");
int age = mInMsg.getInt("age");
```

ObjectMessage

As with `TextMessage`, `ObjectMessage` (Table 3–6) offers a simple interface that supports the insertion or retrieval of objects from the message.

Table 3–6 Accessor Methods for ObjectMessage Interface

`getObject()`	`setObject(Object value)`

Objects passed must implement the `java.io.Serializable` interface:

```
/*Sender*/
//create and send ObjectMessage
String firstName = "JOHN";
String lastName = "DOE";
int age = 33;
Person pObj = new Person(firstName, lastName, age);

ObjectMessage objOutMsg = session.createObjectMessage();
objOutMsg.setObject(pObj);
sender.send(objOutMsg);

/*Receiver*/
//receive and unpack ObjectMessage
ObjectMessage objInMsg = (ObjectMessage)receiver.receive();
Person pObj = (Person)objInMsg.getObject();
```

Summary

We began our exploration of the JMS message by examining the concepts and considerations associated with defining messages. We discussed how message content is defined and reviewed three popular physical formats used to structure message content: XML, tagged/delimited, and record-oriented. Physical formats enable the application to make sense of the data, and for each format we reviewed its history and detailed how it could be used by the JMS client.

We then examined the structure of the JMS message and detailed the attributes that comprise the header, properties, and body. We considered how some of these attributes could be used to effect application processing and specifically devoted some time to examining the use of message selectors to control which messages are delivered to the JMS client. We also discussed which types of JMS messages are suited to various enterprise messaging scenarios, concluding that `TextMessage` and `BytesMessage` are the most flexible and versatile.

The chapter concluded with a detailed review of the accessor methods defined by each type of message interface, and using code snippets, we demonstrated how a JMS client creates, packs, or unpacks a `BytesMessage`, `TextMessage`, `StreamMessage`, `MapMessage`, and `ObjectMessage`.

In the following chapter we examine how our JMS message can be sent or received using the JMS API.