CHAPTER 1

JDBC Today

INTRODUCTION

Since its inception in 1995 the Java language has continued to grow in popularity. Originally intended as a language for embedded systems, the Java language has moved far beyond that. Today Java is used by millions of developers in a myriad of development efforts, from distributed components such as Enterprise JavaBeans, to client-side GUI development with Swing and AWT. Java is used to create Web pages with Java Server Pages and servlets, and to develop Web applications with Java plug-in applets and Java Webstart.

A common thread running through these applications is that they all need data. As the marketing message of the Internet age constantly reminds us, information drives the enterprise. That information is consumed by applications, and the Java Database Connectivity (JDBC) API represents the tool of choice for Java applications to access that data.

JDBC DESIGN

Just as Java was designed to provide platform independence from hardware/software platforms, so too JDBC has been designed to provide some degree of database independence for developers. JDBC is designed to provide a *database-neutral* API for accessing relational databases from different vendors. Just as a Java application does not need to be aware of the operating system platform on which it is running, so too JDBC has been designed so that the database application can use the same methods to access data regardless of the underlying database product.

JDBC was developed to work with the most common type of database: the relational database. This is not to say that JDBC cannot be used with another type of database. In fact, there are JDBC drivers that allow the API to be used to connect to both high-end, mainframe databases, which are not relational, and to access flat files and spreadsheets as databases (which are definitely not relational). But the reality is that JDBC is most commonly used with relational databases.

THE RELATIONAL DATABASE

The technical definition of a relational database is a database that stores data as a collection of related *entities*. These entities are composed of *attributes* that describe the entity, and each entity has a collection of *rows*. Another way to think about a relational database is that it stores information on real-world *objects* (the entities). The information about the objects is contained in the *attributes* for the object.

Since real world objects have some type of relation to each other, we must have a facility for expressing relations between the objects in the database. The relationships between the database objects is described using a query language, the most popular of which is the Structured Query Language (SQL). (Chapter 2 will describe relational databases and SQL in more detail.)

The relational database is the predominant form of database in use today. Other database types include hierarchical, network, flat-file databases, and object databases. Though the hierarchical database is still common on many mainframe systems, it is not commonly used on other platforms.

JAVA AND RELATIONAL DATABASES

Since Java is an object-oriented language, it does not manage data as a relational database does. Data is modeled as objects in Java application design. These objects contain *attributes* (also referred to as *members*), which represent the details of the object. From an object design perspective, an object is not stored—it persists. Its life

extends over multiple invocations of the application. These objects are manipulated using a procedural language with syntax similar to the C programming language.

All of this differs markedly from relational databases, which represent data with tables and columns and manipulate the data using the non-procedural SQL. What we are left with is an impedance mismatch between the object-oriented model of Java and the relational model of relational databases. Ultimately, we must reconcile this difference with our class design. This is a process known as object-relational (OR) mapping and can be done manually by applying certain design patterns, as we do later in this text, or can use various OR-mapping tools (for example, TopLink - `www.objectpeople.com`, CocoBase - `www.thought-inc.com`).

OBJECT DATABASES

From a pure object-oriented perspective, object databases provide a nice fit for object-oriented development. Object databases are available that provide APIs and query languages that can be used for Java. While these tools do provide a convenient facility for persisting objects in Java, they generally do not provide a standardized query language, and they begin to experience performance issues as the size of the data set grows larger and queries against the data become more complex.

Object database management systems (ODBMS) have always enjoyed some degree of popularity in some information technology (IT) sectors—for example, finance and research—but for various reasons, these databases do not currently enjoy the popularity of relational databases.

OBJECT-RELATIONAL MAPPING TOOLS AND JDO

An interesting alternative to both JDBC and object databases is the *Java Data Object* (JDO) API. JDO provides a vendor-neutral facility for persisting Java objects. Like the object database, this represents a natural, object-oriented approach to working with data in a Java application. Issues such as transaction support and query language capabilities are provided for in the JDO specification. Since the JDO specification is not specific to any vendor, a developer could create a Java application using JDO with SQL-Server and port it to use Oracle or DB2 without needing to change any code.

JDO is not necessarily a replacement for JDBC but is instead a complementary approach. JDO will provide for the OR-mapping between the object definitions of Java and the entities and attributes of the relational database, and JDBC will provide the low-level access to the database. JDO and JDBC could be used

together in an application with JDO being used to manage a large number of persistent objects and JDBC being used to provide access to complex, legacy relational databases that prove too difficult and expensive to map into objects.

Limitations of OR-Mapping and JDO

On the surface, JDO and OR-mapping provide a very attractive approach, but there are potential issues. Data queries can become very complex even for a relatively simple application. The nonprocedural nature of SQL allows complex queries to be expressed relatively simply. It remains to be seen whether or not the query language of JDO will provide this expressive elegance.

There is significant technology and experience that relational databases have accumulated over the past 20 or more years that provide performance and usability benefits for the application developer. Additionally, a significant amount of existing data that Java applications must access is in relational databases.

RELATIONAL DATABASES AND SQL

One of the major benefits of relational databases is that they virtually all use standard SQL for a query language. Initially it was hoped that with SQL, applications that were developed to work with a database from one vendor could easily be ported to work with a database from another vendor. But that has not been the case. Database vendors, in an effort to distinguish themselves from one another, have extended the SQL language in many ways.

The extensions to SQL have been both problematic and beneficial. They have been problematic in that a standard was being extended by vendors and thus reducing the benefit of having a standard. But they have been a benefit in that the extensions were often very useful (Oracle's `decode` statement, for example).

Part of the extensions to existing SQL implementations are the Stored Procedure Languages (SPL). Since SQL is a nonprocedural language, it has difficulties managing certain complex operations where many layers of logic must be applied, such as applying complex business rules to large amounts of report data. SPLs are procedural languages like C or Java and can manage these complex logical operations by providing procedural language facilities, such as conditional statements and flow of control operators, and the ability to declare methods or functions.

These SPL implementations are complete programming languages that are implemented within the database engine. It may seem that the inclusion of a programming language in the database engine is redundant and unnecessary when we are working with a full-fledged programming language like Java. But the advantage of using an SPL to perform data processing is that the processing is done in the database engine. The data used in the SPL procedure resides in the

memory space of the database engine, so there is no need to move the data across the network to a program in order to perform the processing. While this performance advantage may not be significant for the processing of 2,000 small rows of data, it does become significant where large pools of data are being processed—for example, the processing of a million rows of data. With large blocks of data, the use of an SPL can mean the difference between only 1 hour of processing for a million rows using an SPL procedure and 8 hours of processing required to extract the data from the database and process it within a program.

Many relational databases also provide database *triggers*. These triggers are associated with a database table and initiate various actions when database activity takes place against the table. Database *update triggers,* probably the most common type of trigger, are executed when a database insert, update, or delete is run against a database table. These triggers are an excellent means of enhancing database integrity and can be used to enforce business rules, replicate data, and provide auditing type facilities by logging table updates.

Other important extensions to relational databases include data fragmentation where data for a table is distributed across separate logical devices, thus improving performance for scans of a large number of rows from the table. Also, database replication where two different database servers running on two different machines remain completely synchronized provides significant benefits.

THE JDBC API

The JDBC API was released in 1997 following a series of specifications that were finalized in the previous year. The API was designed to make the Call Level Interface (CLI) access of relational databases vendor-neutral. Each relational database vendor had created its own version of a CLI for accessing its database. These CLIs were primarily created for the C programming language and later C++. To reduce confusion over these varying CLI implementations, the X/Open Consortium created a standard CLI specification.

JDBC is currently divided into two Java packages: `java.sql` and `javax.sql`. The `java.sql` package contains the core of the original JDBC API and the various improvements on that package that have been made over the years. The `javax.sql` package contains the extensions to the JDBC API that provide some very useful features that were originally added as part of the JDBC 2.0 standard extensions (yes, a contradiction in terms). Both the `java.sql` package and `javax.sql` package are part of the J2SE 1.4 release.

The JDBC specification provides a set of interfaces that database vendors must implement. Vendors have some flexibility in how they implement the JDBC specification. Four different types of implementations have been identified, as detailed in Table 1–1.

Table 1–1 *JDBC Driver Types*

Driver	Description
Type 1	Implements JDBC by mapping JDBC calls to other CLI calls. Uses a binary library written in another language. Requires software on the client machine, for example, the JDBC-ODBC bridge driver.
Type 2	Driver is partially composed of Java code and partially in native code using another CLI . Requires some client-side binary code.
Type 3	Pure Java driver; uses middleware to convert JDBC calls to vendor-specific calls and protocol required to access the database.
Type 4	Pure Java driver that implements the native protocol. Does not require middleware or any client-side binary. Can be downloaded to a client if necessary.

There are four different types of JDBC drivers. The distinctions between these drivers are based primarily on the components of the driver, where the components must reside, and the language used to develop the components. Each database vendor uses a different set of calls and a different network protocol to access its database. These database vendors offer their own proprietary APIs and drivers to provide access to their databases, and with all JDBC driver types, JDBC calls must be *mapped* or converted to the vendor protocol. In the case of the Type 1 driver, this mapping has an additional layer of indirection through the binary library to the native CLI. The Type 3 driver provides this mapping through a middleware server component that communicates with the client-side driver and provides mapping and database communication. The Type 4 driver provides this mapping through pure Java code written to manage the vendor-specific protocol.

Type 1 and Type 2 drivers require binaries on the client machine. Type 3 and Type 4 drivers, however, are pure Java solutions that significantly reduce porting issues for JDBC driver providers.

Type 2 drivers require some binary code to reside on the client machine. JDBC calls are converted into vendor-specific protocol for the database vendor, potentially mapping the calls to a database driver (usually provided by the database vendor) written in some other language.

The type of driver generally recommended is the Type 4 driver. The fact that it is pure Java code enhances portability, which means driver developers are not stretched thin supporting multiple ports. The Type 4 driver also enjoys potential

performance benefits from more efficient code, since JDBC calls do not need to be mapped to proprietary CLI calls (as in Type 2 drivers) and there is no middleware to add additional network overhead, as with Type 3 drivers.

PROGRAMMING FOR TODAY

Today's programming goes far beyond the simple needs of client-server or monolithic applications. In the age of the Internet, it is not unusual and is often necessary for an application to be composed of many different parts or *components* spread across multiple machines. This distributed programming requires a *multitiered* or *n-tiered* development approach.

Multitiered programming is also known as *distributed programming:* an application that is composed of multiple components working together. These multiple components may run on one server or on many servers—they are still collectively considered a complete application.

With this approach, a single application is composed of multiple components running on distinct architectural *tiers*. From a design perspective, the composition of these logical tiers, the work that will be performed on these tiers, should reflect the "responsibility" of the components. This benefits the development effort by providing a consistent structure to multitiered applications. A common approach to n-tiered development uses the following tiers.

- client tier
- presentation tier
- business tier
- resource tier

The *client tier* is responsible for interacting with the user. This interaction will include the display (or rendering) of the user interface and the initial processing of user input. In a Web application, the client tier is the Web browser.

The *presentation tier* is responsible for preparing the output to the client tier and interfacing with the business tier. The presentation tier should not execute any business logic. That is, it should not enforce the business rules of the enterprise; that work should be left to the business tier. In a Web application, the presentation tier is usually a Web server with the ability to process JSP or servlet pages.

The *business tier* is responsible for the execution of the business logic of the enterprise. This tier is expected to process requests from the presentation tier: requests that have been forwarded from the client tier. The business tier will interface with the resource tier to obtain the data that it needs to complete its processing.

The *resource tier* is responsible for managing the resources of the application. For most applications, this tier represents the database. This is where the application data that will persist will be stored and managed.

JDBC Code in N-Tiered Architectures

Java code using JDBC usually resides on the business tier. The code performing the data access on this tier should be isolated and encapsulated in a set of *black box* objects, objects which conceal their details and expose a concise interface.

Multitiered/distributed application architecture and Java design patterns will be explained in more detail in Chapter 13. What is worth noting at this point is that JDBC code will be used differently depending on the component we are writing and the architectural tier where that component will be placed. We can use Java design patterns to help guide this coding process.

Java Technologies for Distributed Programming

Sun Microsystems has packaged a number of Java technologies together under the marketing and distribution umbrella of the Java 2 Enterprise Edition (J2EE). This package is comprised of numerous APIs and technologies that represent the Java tools for developing distributed applications. To develop a Web application using Java technology, these are the tools to use. The core of J2EE includes the APIs listed in Table 1–2 (which include JDBC).

J2EE not only includes Java APIs, but requires servers to run the various components created using the Java APIs. For instance, servlets and JSP pages must run with a servlet server that provides what is known as a *servlet container*. The Java applet must in turn run within what is known as an *applet container*. Enterprise Java Beans (EJB) must run within an *EJB container*. J2EE and the technology behind it is covered in more detail in Chapter 22 and Chapter 23.

Sun has expanded on its J2EE architecture, in a large part in response to the overwhelming interest (whether justified or not) in *Web services*. A Web service is a service that makes itself available over the Internet using HTTP and involves the exchange of messages in XML format. Web services have been popularized to a large extent by Microsoft, which has made Web services a key part of its .Net architecture. Sun has expanded and refined its J2EE architecture to include additional services in what has been dubbed the Open Net Environment (ONE).

J2EE is a distributed component architecture and, as such, does not limit components to the exchange of messages in XML format. Instead, components can communicate with a variety of protocols, including the binary protocols of RMI-IIOP, and also including the asynchronous message passing of JMS or Message Beans and participation in SOAP transactions if the application server provides that. So, while not part of the current J2EE specification, Web services are part of what Sun considers a valid enterprise architecture.

Table 1–2 *J2EE APIs*

API/Java Technology	Description
servlets, JSP	Distributes HTML output over HTTP connection. JSP Java Server Pages extends the servlet API and includes a preprocessor that converts a JSP page into a servlet, which is then run in a servlet engine.
EJB	Enterprise Java Beans. A distributed component technology that provides a number of standard services, such as persistence, transactions, security, and others.
JMS	Java Messaging Service. Common access to message servers for asynchronous message communication.
JDBC	Provides communication with relational databases.
JavaMail	Access to POP3, IMAP, and other standardized mail servers.
JNDI	Java Naming and Directory Interface. Used to provide general lookup of objects and application properties.
JAF	Java Activation Framework. Used with JavaMail for viewing/editing of MIME content.
Java–IDL	Provides access to CORBA components using Java.
RMI, RMI-IIOP	Remote Method Invocation. Provides the ability to create remote objects and execute methods (passing parameters and receiving return values) with those remote objects.
JTS/JTA	Java Transaction API. An API that provides access to transaction controls. Uses Java Transaction Service as the low-level implementation (the service provider interface) for the transaction service.
Java-XML	XML-encoded documents provide much of the configuration information for J2EE. XML is becoming more important as a means of data encoding for data interaction. Provides parsers (JAXP), messaging (JAXMP), registries (JAXR) and RPC (JAX-RPC) APIs.

JAVA DESIGN PATTERNS

The concept of design patterns is often heard discussed in connection with Java application design. Design patterns are used to help guide the development process. Design patterns do not represent complete, template-like solutions, but instead represent recommendations on how to solve certain recurring problems with Java development.

The concept of design patterns can be traced back to work that Christopher Alexander did with building construction architecture in the 1980s. Alexander noted that certain problems would consistently recur in building design and that certain proven solutions could be used to solve these problems. He referred to these proven solutions as *design patterns*.

A group of academics picked up on this work and wrote a seminal book on the subject titled, appropriately enough, *Design Patterns: Elements of Reusable Object-Oriented Software*. The authors of this text, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, are often referred to as the Gang of Four, and thus the text is often referred to as the GoF book. The authors of this text very succinctly applied the concept of design patterns to the process of developing good, object-oriented code. They used the Smalltalk language, but the solutions can easily be applied to any full-featured object-oriented language that supports polymorphism and a facility similar to Java interfaces.

Chapter 13 covers design patterns that apply to JDBC programming in more detail. What is important to note at this point is that design patterns have a significant impact on how JDBC will be used in an application. It should also be noted that design patterns can be applied at several different levels of the development process. Gamma, Helm, Johnson, and Vlissides refer to design patterns as taking the form of creational, structural, or behavioral patterns in relation to how the patterns will be used.

A design pattern that is often noted is the Model, View, Controller (MVC) design pattern. The MVC design pattern was originally applied to the Graphical User Interface (GUI) programming and describes the responsibilities of different portions of the application, as shown in Table 1–3.

As applied to a GUI application, the *model* portion of the application manages the data, the *view* displays the controls of the application (input fields, tables, list boxes), and the *controller* represents the event handlers for user-generated events: button clicks, list box choices, and others. In a GUI application being developed with an object-oriented language, these components would represent objects (and the class definitions for the object) that would be designed to provide for the behaviors, the responsibilities described in Table 1–3.

Table 1–3 *MVC Design Pattern*

Component	Responsibility
Model	Manages the application state, the data the application is using.
View	Renders the portion of the application visible to the user.
Controller	Responds to user gestures and interfaces with the model to control the application.

Table 1–4 *J2EE Components by Tier*

Tier	MVC	Component
Client	View	Applet, HTML, Java Webstart
Presentation	Controller	servlet, JSP page
Business	Model	Enterprise JavaBean, JavaBean
Resource	n/a	SQL

But in order to apply this design pattern to a Web application, it is imperative that we identify which Web application components will implement the design. If we approach this design pattern using our multitiered architecture described earlier, we should expect that the view portion will be managed by the client tier, the controller will be managed the presentation tier, and the model will be managed by the business tier components. If we are using J2EE, our most likely candidate for each of these components is as follows (Table 1–4).

When viewed in this respect, the MVC pattern describes the responsibilities of the components being used and so would probably more accurately be described as an *architectural* design pattern. The MVC pattern alone does not describe how the specific components (the view component being used on the presentation tier, for example) would be designed. Other Java design patterns as shown later in this book do provide these details.

Using MVC as an architectural pattern, we do receive some high-level guidance about where JDBC code would be located. We would expect the JDBC calls to be placed in the components in the business tier. Located in that tier, the JDBC calls would retrieve data from the resource tier, and the Java code in the business tier would apply business logic and then return the data to the presentation tier, where it would be formatted for presentation to the user.

SUMMARY

This chapter introduced the topic of Java database programming in today's programming world. We have seen how Java has grown and progressed from a small side project at Sun Microsystems to the language of choice for enterprise application development. We have also seen how JDBC fits into this picture as being the API developed by Sun to provide access to relational databases using a vendor-neutral API.

There are some alternatives to JDBC, such as OR-mapping and the incipient JDO. But when existing relational databases with all their complexity must be accessed, JDBC is the tool of choice.

COMING UP NEXT

Chapter 2 examines the target of most JDBC applications: the relational database. The roots of the relational database are covered, and an introduction to the lingua franca of the query languages—SQL—will be provided. Following that chapter, we will begin our detailed discussion of the JDBC API.