# Part One

Part One contains material that introduces JDBC technology. The first chapter, "Introduction," briefly explains what JDBC technology is and presents overviews of the Java programming language and relational databases. The four tutorial chapters demonstrate how to use the JDBC API. "Basic Tutorial" walks you through the basic API, "Advanced Tutorial" shows how to use the functionality introduced in the JDBC 2.0 API and in the JDBC 3.0 API. "Metadata Tutorial" explains how to use the metadata API, and "Rowset Tutorial" demonstrates what you can do with rowsets.
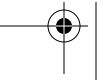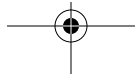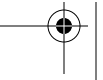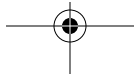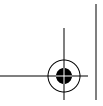
C H A P T E R 1

Introduction

THIS book covers all of the JDBC™ API, the application programming interface that provides universal data access for the Java™ programming language. The first edition covered the JDBC 1.0 API, which provides the basic functionality for data access. The second edition added the JDBC 2.0 API, which supplements the basic API with more advanced features. The 2.0 API provides a standard way to access the latest object-relational features being supported by today's relational database management systems. In addition, it includes features such as scrollable and updatable result sets and improved performance. Further, it extends JDBC technology beyond the client to the server with connection pooling and distributed transactions.

   This third edition covers the complete JDBC 3.0 API. The specification for the JDBC 3.0 API combines all previous specifications, including the JDBC Optional Package specification, into one comprehensive document. It also adds new functionality, such as savepoints, auto-generated keys, and parameter metadata, plus enhancements to existing functionality. Beginning with the release of the Java™ 2 Platform, Standard Edition (J2SE™), version 1.4, the complete JDBC 3.0 API is bundled as part of the J2SE download. Previously, the JDBC Optional Package had to be downloaded separately.

   A summary and complete list of the features added in the JDBC 2.0 API and JDBC 3.0 API can be found in Appendix B, starting on page 1121. An application will run successfully only if the driver and DBMS support all of the functionality it uses.

1.1   What the JDBC 3.0 API Includes

The JDBC 3.0 API includes all of the API in the `java.sql` package (sometimes called the core API) and the `javax.sql` package (the Optional Package API, formerly called the Standard Extension API).

The following list defines terms as they are used in this book.

- JDBC 3.0 API—the incorporation of all previous JDBC specifications plus the addition of new functionality.

- JDBC 2.0 API—the complete JDBC 2.0 API, including both the `java.sql` package (the JDBC 2.1 core API) and the `javax.sql` package (the JDBC Optional Package API).

- JDBC 2.0 core API—the JDBC 2.0 `java.sql` package. Some of the features introduced in this package are scrollable result sets, batch updates, programmatic updates, and support for the new SQL99 data types. For changes between JDBC 2.0 core API and JDBC 2.1 core API, see "Overview of JDBC 2.0 Core API Changes," on page 1132. In this book, "JDBC 2.0 core API" is a generic term that includes the JDBC 2.1 core API.

- JDBC Optional Package API—the package `javax.sql`. This package makes it easier to build server-side applications using the Java platform by providing an open architecture that supports connection pooling and distributed transactions that span multiple database servers. The `DataSource` API plays an integral part in these capabilities and also works with the Java™ Naming and Directory Interface™ (JNDI) to improve portability and make code maintenance easier. The `javax.sql` package also provides the `RowSet` API, which makes it easy to handle data sets from virtually any data source as JavaBeans™ components.

- `java.sql` package—the core JDBC API, including the JDBC 1.0, JDBC 1.1, JDBC 2.0, JDBC2.1, and JDBC 3.0 API.

- `javax.sql` package—the JDBC Optional Package API

The earliest book about the JDBC API in the Java Series, *JDBC™ Database Access with Java™*, covered only the JDBC 1.0 API. The second book, *JDBC™ API Tutorial and Reference, Second Edition,* built on the earlier one, updating the original material where necessary and adding a great deal of new material. In similar fash-

ion, this book, *JDBC^{TM} API Tutorial and Reference, Third Edition*, builds on the second edition, adding new material and revising existing material where necessary.

In keeping with the policy of maintaining backward compatibility, applications written using the JDBC 1.0 API will continue to run with both the Java 2 SDK, Standard Edition, and the Java 2 SDK, Enterprise Edition, just as they have always run. Having been well designed from the beginning, the JDBC 1.0 API is essentially unchanged. Applications using features added in the JDBC 2.0 API or 3.0 API will, of course, need to be run using a driver that supports those features.

1.2   Conventions Used in This Book

This book uses various conventions as aids to understanding or as a means of reducing repetition.

1.2.1   Fonts to Indicate Function

Different fonts indicate that text is being used in a special way.

FONT	USED FOR
`Lucida sans Typewriter`	code, which includes what would be typed in a source code file or at the command line; URLs; file names; keywords; name of a class, interface, exception, constructor, method, or field
italic code font	a variable used in text or in a method explanation
italic	a new term being introduced; emphasis
➥	the output of executing JDBC code

1.2.2   Icons to Indicate New Material

Some readers will be using drivers and data sources that do not yet implement the new features in the JDBC 3.0 API. Some will be using drivers that do not implement the JDBC 2.0 API. To make it easy to see what has been added to the JDBC 1.0 API, new material is marked with an icon to indicate when it was added.

New features in the JDBC 3.0 core API are marked with the 3.0 icon:

3.0

Features added in the JDBC 2.0 core API are marked with the 2.0 icon:

2.0

The JDBC Optional Package features are marked with the `javax.sql` icon:

〈javax.sql〉

Note that the `javax.sql` icon indicates API that was added in the 2.0 time frame. As of the JDBC 3.0 API specification, both the JDBC 2.0 core API and the JDBC Optional Package API are part of the JDBC 3.0 API. The icons indicate when the API was added.

   If an entire reference chapter is new, it will have an icon to the right of the chapter title to indicate when it was introduced. New chapters also have an icon in the outer margin by the class or interface definition heading to indicate that all the API listed is new. For convenience, all explanations of new methods, constructors, and fields are individually marked with an icon so that someone looking up only a method or field can see whether it is new in the JDBC 2.0 API or the JDBC 3.0 API without checking for an icon at the beginning of the section.

   In chapters where only some material is new, an icon will appear in the margin to indicate what is new. If an entire section is new, the icon will appear next to the section heading; an icon next to a paragraph indicates that part of the section, starting at the icon, is new. In the class or interface definition section and in the sections explaining constructors, methods, exceptions, or fields, an icon marks the API that is new.

   Appendix B, "Summary of Changes," starting on page 1121, summarizes all of the new features and gives a complete list of what has been added to the JDBC API. The summary is divided into three sections: one for new features in the JDBC 3.0 API, one for the features added in the JDBC 2.0 core API, and another section for the Optional Package API.

1.2.3   Special Page Designations in the Index

The index uses "T" after a page number to indicate that material is in a tutorial. The designation "tb" after a page number means that the information is in a table.

1.2.4 `SQLException` Is Implied in Method Explanations

In Part Two, every method whose signature includes "`throws SQLException`", which is nearly every method in the JDBC API, may throw an `SQLException`. In the overwhelming majority of cases, this exception is thrown because there has been an error in attempting to access data. Access errors can be caused by a locking conflict, a deadlock, a permission violation, a key constraint (trying to insert a duplicate key, for instance), and so on.

To conserve space, one explanation of `SQLException` is given here rather than being repeated in nearly every method explanation throughout the reference section. In other words, the fact that a method throws an `SQLException` when there is an access error is implied for each and every method whose signature includes "`throws SQLException`." The `Throws` section in a method explanation is implied and not included when the only reason for throwing the `SQLException` is an access error. If an `SQLException` is thrown for any other reason, a `Throws` section gives the exception and the conditions that cause it to be thrown. Other exceptions that a method can throw are always listed and explained.

1.2.5 Some Method Explanations Are Combined

In order to avoid unnecessary repetition, the method explanations for some methods are combined.

- The getter methods and the updater methods in the `ResultSet` interface all have two basic versions, one that takes a column index as a parameter and one that takes a column name as a parameter. The explanations for both versions are combined into one entry. This also applies to the getter methods in the `CallableStatement` interface.

- Every getter and updater method in the `ResultSet` interface takes a column parameter. This parameter (a column index or a column name) is explained only once, at the beginning of the section "`ResultSet` Methods," on page 730. The explanation is implied for every `ResultSet` getter method and for every `ResultSet` updater method rather than being repeated for each method.

- The three versions of the `DriverManager.getConnection` method are combined into one entry.

1.3   Contents of the Book

As was true with the previous editions, this book is really two books in one: a tutorial and the definitive reference manual for the JDBC API. The goal is to be useful to a wide range of readers, from database novices to database experts. Therefore, we have arranged the book so that information needed only by experts is separate from the basic material. We hope that driver developers and application server developers as well as application programmers and MIS administrators will find what they need. Because different sections are aimed at different audiences, we do not expect that everyone will necessarily read every page. We have sometimes duplicated explanations in an effort to make reading easier for those who do not read all sections.

1.3.1   Part One

Part One includes five chapters, an introduction and four tutorial chapters. Part Two, the reference manual, has a chapter for each class or interface, a chapter on the `DISTINCT` data type, a chapter on how SQL and Java types are mapped to each other, two appendices, a glossary, and an index.

Chapter 1, "Introduction," outlines the contents of the book and gives overviews of the Java programming language and SQL. The overview of the Java programming language summarizes many concepts and is not intended to be complete. We suggest that anyone who is unfamiliar with the language refer to one of the many excellent books available. The overview of relational databases and SQL likewise covers only the highlights and presents only basic terminology and concepts.

Chapter 2, "Basic Tutorial," walks the reader through how to use the basic JDBC API, giving many examples along the way. The emphasis is on showing how to execute the more common tasks rather than on giving exhaustive examples of every possible feature.

Chapter 3, "Advanced Tutorial," starting on page 113, covers the new features added to the `java.sql` package in the JDBC 3.0 API and the JDBC 2.0 core API and also some of the features from the `javax.sql` package. Although the data types and new functionality can be considered advanced in relation to the material

in the basic tutorial, one does not need to be an advanced programmer to learn how to use them. On the contrary, one of the strengths of the new JDBC API is that it makes using the new features easy and convenient.

Chapter 4, "Metadata Tutorial," shows how to use the JDBC metadata API, which is used to get information about result sets, databases, and the parameters to `PreparedStatement` objects. It will be of most interest to those who need to write applications that adapt themselves to the specific capabilities of several database systems or to the content of any database.

Chapter 5, "Rowset Tutorial," gives examples of how to use rowsets and summarizes the `RowSet` reference implementations that are being developed to serve as standards for further implementations.

1.3.2   Part Two

Part Two is the definitive reference manual for the complete JDBC 3.0 API. Thus it covers both the core JDBC API (the `java.sql` package) and the Optional Package JDBC API (the `javax.sql` package).

Chapters 6 through 50, which are arranged alphabetically for easy reference, cover all of the JDBC classes and interfaces. The only chapter that does not refer to a class or interface is Chapter 19 on the JDBC type `DISTINCT`; this data type maps to a built-in type rather than to a JDBC class or interface.

Chapter overviews generally show how to create an instance of the class or interface and how that instance is commonly used. Overviews also present a summary of what the class or interface contains and explanatory material as needed.

The class and interface definitions list the constructors, methods, and fields, grouping them in logical order (as opposed to the alphabetical order used in the sections that explain them).

Sections for explanations of each constructor, method, and field follow the class or interface definition. The explanations in these sections are in alphabetical order to facilitate looking them up quickly.

Chapter 50, "Mapping SQL and Java Types," which explains the standard mapping of Java and SQL types, includes tables showing the various mappings.

Appendix A, "For Driver Writers," contains information for driver writers, including requirements, allowed variations, and notes on security.

Appendix B, "Summary of Changes," summarizes the new features in the JDBC 3.0 API and the JDBC 2.0 API. It presents a complete list of every new interface, class, exception, constructor, method, and field. The summary for the JDBC 3.0 API comes first, followed by summaries of the JDBC 2.0 core API and

the JDBC Optional Package API. This is followed by a list of the API that has been deprecated, which includes the API to use in place of the deprecated API, where applicable. The last part of Appendix B gives a brief history of API changes, going back to the beginning, and also explains various early design decisions. This section should answer some questions about how JDBC got to its present form.

Completing the book are a glossary and a comprehensive index, which we hope readers find helpful and easy to use.

A Quick Reference Card can be found inside the back cover. It includes the most commonly used methods and SQL/Java type mappings.

1.3.3   Suggested Order for Reading Chapters

This section suggests an order in which to read chapters for the person learning how to use the JDBC API. There is no right or wrong way, and some may find another order better for them. For example, some people may find it more helpful to read relevant reference chapters before they look at the tutorials. Some may prefer to read only the tutorials. The only firm suggestions are that beginners should read Chapters 1 and 2 first and that everyone should read the section "Conventions Used in This Book," on page 5. The following is one suggested order for reading chapters:

1. Chapter 1, Introduction

2. Chapter 2, Basic Tutorial

   ▪ ResultSet

   ▪ Chapters on SQL statements

     • Statement
     • PreparedStatement
     • CallableStatement

   ▪ Connection

   ▪ Chapters on establishing a connection

     • DriverManager
     • DataSource

3. Chapter 3, Advanced Tutorial

- Mapping SQL and Java Types

- SQL99 Types

  - Clob
  - Blob
  - Array
  - Struct
  - Ref

- Exceptions

  - SQLException
  - BatchUpdateException
  - SQLWarning
  - DataTruncation

4. Chapter 5, Rowset Tutorial

- Rowset

5. Chapter 4, Metadata Tutorial

- ResultSetMetaData

- DatabaseMetaData

- ParameterMetaData

1.3.4   Where to Find Information by Topic

This section groups chapters together by topic to make it easier to find information about a particular topic. Each topic gives the chapter to read for an overall explanation, the tutorial in which the topic is illustrated, and a list of related chapters. The Index is the place to look for more specific topics.

Executing SQL statements
- Overall explanation: Statement
- Tutorial: Basic Tutorial
- Related chapters: PreparedStatement, CallableStatement

Batch updates
- Overall explanation: Statement

- Tutorial: Advanced Tutorial
- Related chapters: PreparedStatement, CallableStatement, BatchUpdate-Exception

Custom mapping
- Overall explanation: Struct, SQLData
- Tutorial: Advanced Tutorial
- Related chapters: SQLInput, SQLOutput
- Explanation of type maps: Connection

Rowsets
- Overall explanation: RowSet
- Tutorial: Rowset Tutorial
- Related chapters: RowSetEvent, RowSetListener, RowSetInternal, RowSetMetaData, RowSetReader, RowSetWriter

Connection pooling
- Overall explanation: PooledConnection
- Tutorial: Advanced Tutorial
- Related chapters: ConnectionPoolDataSource, ConnectionEvent, Connection Event Listener, DataSource

Transactions
- Overall explanation: Connection
- Tutorial: Basic Tutorial

Distributed transactions
- Overall explanation: XAConnection
- Tutorial: Advanced Tutorial, Rowset Tutorial
- Related chapter: DataSource, XADataSource

Date-related types
- Overall explanation: Date
- Related chapters: Time, Timestamp

User-defined types
- Overall explanation: Struct, Distinct
- Tutorial: Advanced Tutorial
- Related chapters: See custom mapping chapters

1.3.5   Resources on the Web

This section lists the URLs for getting information about the JDBC API and related technologies.

- JDBC 3.0 specification, JDBC 1.0 API specification, JDBC 2.1 Core API specification, and JDBC 2.0 Optional Package API specification

  `http://java.sun.com/products/jdbc`

- Java Transaction API (JTA)

  `http://java.sun.com/products/jta`

- Java Transaction Service (JTS)

  `http://java.sun.com/products/jts`

- Java Naming and Directory Interface (JNDI)

  `http://java.sun.com/products/jndi`

- Enterprise JavaBeans (EJB)

  `http://java.sun.com/products/ejb`

1.4   What Is the JDBC API?

The JDBC API is a Java API for accessing virtually any kind of tabular data. (As a point of interest, JDBC is a trademarked name and is not an acronym; nevertheless, JDBC is often thought of as standing for "Java Database Connectivity." Originally, JDBC was the only trademarked name for the data source access API, but more recently, Java™ DataBase Connectivity has been added as a second trademarked name.) The JDBC API consists of a set of classes and interfaces written in the Java programming language that provide a standard API for tool/database developers and makes it possible to write industrial-strength database applications entirely in the Java programming language.

The JDBC API makes it easy to send SQL statements to relational database systems and supports all dialects of SQL. But the JDBC API goes beyond SQL, also making it possible to interact with other kinds of data sources, such as files containing tabular data.

The value of the JDBC API is that an application can access virtually any data source and run on any platform with a Java Virtual Machine. In other words, with the JDBC API, it isn't necessary to write one program to access a Sybase database, another program to access an Oracle database, another program to access an

IBM DB2 database, and so on. One can write a single program using the JDBC API, and the program will be able to send SQL or other statements to the appropriate data source. And, with an application written in the Java programming language, one doesn't have to worry about writing different applications to run on different platforms. The combination of the Java platform and the JDBC API lets a programmer "write once and run anywhere™." We explain more about this later.

The Java programming language, being robust, secure, easy to use, easy to understand, and automatically downloadable on a network, is an excellent language basis for database applications. What is needed is a way for Java applications to talk to a variety of different data sources. The JDBC API provides the mechanism for doing this.

The JDBC API extends what can be done with the Java platform. For example, the JDBC API makes it possible to publish a web page containing an applet that uses information obtained from a remote data source. Or, an enterprise can use the JDBC API to connect all its employees (even if they are using a conglomeration of Windows, Macintosh, and UNIX machines) to one or more internal databases via an intranet. With more and more programmers using the Java programming language, the need for easy and universal data access from the Java programming language continues to grow.

MIS managers like the combination of the Java platform and JDBC technology because it makes disseminating information easy and economical. Businesses can continue to use their installed databases and access information easily even if it is stored on different database management systems or other data sources. Development time for new applications is short. Installation and version control are greatly simplified. A programmer can write an application or an update once, put it on the server, and then everybody has access to the latest version. And for businesses selling information services, the combination of the Java and JDBC technologies offers a better way of distributing information updates to external customers.

We will discuss various ways to use the JDBC API in more detail later.

1.4.1   What Does the JDBC API Do?

In simplest terms, a JDBC technology-based driver ("JDBC driver") makes it possible to do three things:

1. Establish a connection with a data source

2. Send queries and update statements to the data source

3. Process the results

The following code fragment gives a simple example of these three steps:

```
Connection con = DriverManager.getConnection(
                    "jdbc:myDriver:wombat", "myLogin", "myPassword");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next()) {
    int x = rs.getInt("a");
    String s = rs.getString("b");
    float f = rs.getFloat("c");
}
```

1.4.2   A Base for Other APIs

The JDBC API is used to invoke (or "call") SQL commands directly. It works very well in this capacity and is easier to use than other database connectivity APIs, but it was also designed to be a base upon which to build alternate interfaces and tools. An alternate interface tries to be "user-friendly" by using a more understandable or more convenient API that is translated behind the scenes into the JDBC API. Because the JDBC API is complete and powerful enough to be used as a base, it has had various kinds of alternate APIs developed on top of it, including the following:

1. An embedded SQL for Java

   A consortium including Oracle, IBM, Sun, and others has defined the SQLJ specification to provide an embedded SQL for the Java programming language. JDBC technology requires that SQL statements basically be passed as uninterpreted strings to Java methods. An embedded SQL preprocessor provides compile-time type checking and allows a programmer to intermix SQL statements with Java programming language statements. For example, a Java variable can be used in an SQL statement to receive or provide SQL values. The SQLJ preprocessor effectively translates this Java/SQL mix into the Java programming language with JDBC calls.

   The SQLJ specification is currently evolving to support JDBC 3.0 features such as savepoints, multiple open ResultSet objects, and the DATALINK data type.

2. Technologies for persisting Java objects

   Two Java technologies provide the ability to map Java objects to relational databases: Java^{TM} Data Objects (JDO) API, developed through the Java Community Process as JSR 12, and Enterprise JavaBeans^{TM} (EJB^{TM}) technologies (Container Managed Persistence (CMP) and Bean Managed Persistence (BMP)). With these technologies, data in a data store can be mapped to Java objects, and Java objects can be stored persistently in a data store.

   For example, some of the the most popular JDO implementations use JDBC to map from Java classes to relational database tables. Each row of the table represents an instance of the class, and each column represents a field of that instance. The JDO specification provides a standard API to access the rows and columns of relational databases as if they were native Java objects stored in the database.

   The JDO API provides transparent database access, letting a programmer write code in the Java programming language that accesses an underlying data store without using any database-specific code, such as SQL. To make this possible, a JDO implementation might use SQL and the JDBC API behind the scenes to access data in a relational database. The JDO API also provides a query language, JDOQL, that allows a user to write a query using Java boolean predicates to select persistent instances from the data store. When a relational database is being accessed, these queries are mapped under the covers directly to SQL queries and executed via the JDBC API. EJB technology also provides its own query language, the Enterprise JavaBeans Query Language (EJBQL).

   JDO technology provides a way to persist plain old Java objects. As such , it complements rather than competes with JDBC technology. JDO technology also complements the CMP and BMP technologies, offering an alternative way to persist objects. The difference is that EJB technology uses entity beans to persist business components, whereas JDO technology persists general Java objects. Both can use the JDBC API "under the covers" in their implementations.

   For more information on the JDO API, see

   ```
   http://java.sun.com/products/jdo
   ```

The JDO user community can be reached at

```
http://JDOCentral.com
```

Information about EJB technology is available at

```
http://java.sun.com/products/ejb
```

3. Tools making it easier to use the JDBC API
   As interest in JDBC technology has grown, various tools based on the JDBC API have been developed to make building programs easier. For example, an application might present a menu of database tasks from which to choose. After a task is selected, the application presents prompts and blanks for filling in information needed to carry out the selected task. With the requested input typed in, the application then automatically invokes the necessary JDBC commands. With the help of such a tool, users can perform database tasks even when they have little or no knowledge of SQL syntax.

1.4.3   The JDBC API versus ODBC

Prior to the development of the JDBC API, Microsoft's ODBC (Open DataBase Connectivity) API was the most widely used programming interface for accessing relational databases. It offers the ability to connect to almost all databases on almost all platforms. So why not just use ODBC from the Java programming language?

The answer is that you can use ODBC from the Java programming language, but this is best done with the help of the JDBC API by using the JDBC–ODBC Bridge, which we will cover shortly. The question now becomes, "Why do you need the JDBC API?" There are several answers to this question.

1. ODBC is not appropriate for direct use from the Java programming language because it uses a C interface. Calls from Java to native C code have a number of drawbacks in the security, implementation, robustness, and automatic portability of applications.

2. A literal translation of the ODBC C API into a Java API would not be desirable. For example, Java has no pointers (address variables), and ODBC makes copious use of them, including the notoriously error-prone generic pointervoid *. You can think of JDBC as ODBC translated into a high-level

object-oriented interface that is natural for programmers using the Java programming language.

3. ODBC is hard to learn. It mixes simple and advanced features together, and it has complex options even for simple queries. The JDBC API, on the other hand, was designed to keep simple things simple while allowing more advanced capabilities where required. The JDBC API is also easier to use simply because it is a Java API, which means that a programmer does not need to worry about either memory management or data byte alignment.

4. A Java API like JDBC is needed in order to enable a "pure Java" solution, that is, a solution that uses only Java API. When ODBC is used, the ODBC driver manager and drivers must be manually installed on every client machine. When the JDBC driver is written completely in Java, however, JDBC code is automatically installable, portable, and secure on all Java platforms, from network computers to mainframes.

5. The JDBC 3.0 API includes functionality that is not available with ODBC. For example, ODBC does not support SQL99 data types, auto-generated keys, or savepoints.

In summary, the JDBC API is a natural Java interface for working with SQL. It builds on ODBC rather than starting from scratch, so programmers familiar with ODBC will find it very easy to learn. The JDBC API retains some of the basic design features of ODBC; in fact, both interfaces are based on the Open Group (formerly X/Open) SQL CLI (Call Level Interface). The big difference is that the JDBC API builds on and reinforces the style and virtues of the Java programming language, and it goes beyond just sending SQL statements to a relational database management system.

Microsoft has introduced new APIs beyond ODBC such as OLE DB, ADO (ActiveX Data Objects), and ADO.NET. In many ways these APIs move in the same direction as the JDBC API. For example, they are also object-oriented interfaces to databases that can be used to execute SQL statements. However, OLE DB is a low-level interface designed for tools rather than developers. ADO and ADO.NET are newer and more like the JDBC API and the `RowSet` interface, but they are not pure Java and therefore do not provide portable implementations.

1.4.4   Two-tier and Three-tier Models

The JDBC API supports both two-tier and three-tier models for database access. Figure 1.1 illustrates a two-tier architecture for data access.

Figure 1.1: Two-tier Model
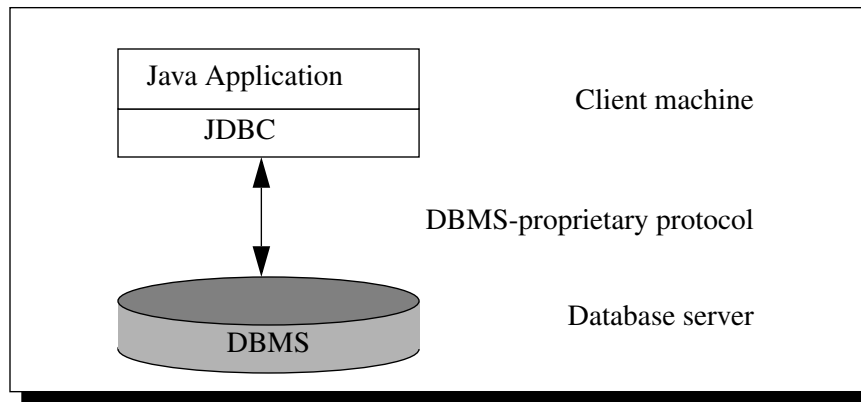
In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a *client/server* configuration, with the user's machine as the client and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

Figure 1.2 illustrates a three-tier architecture for database access.

Figure 1.2: JDBC Three-tier Model
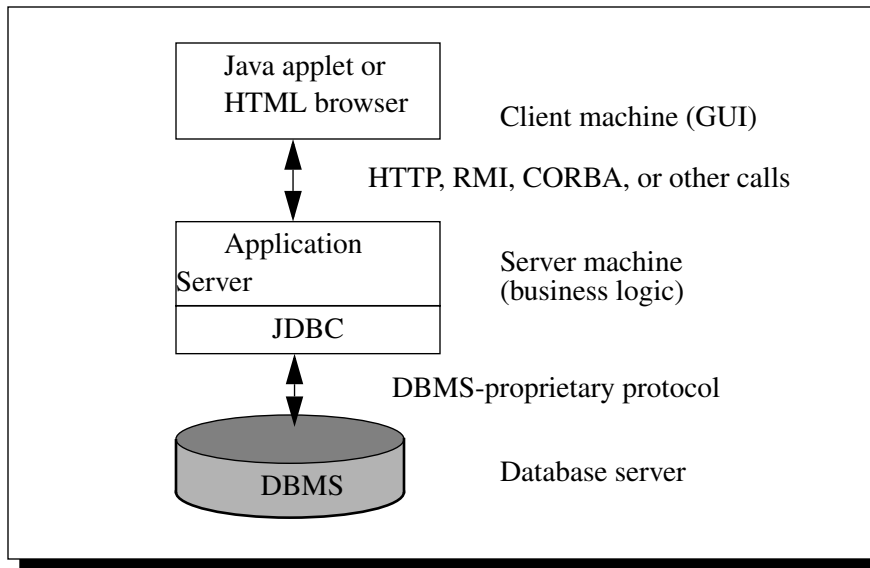
At one time, the middle tier was typically written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and the widespread adoption and success of the J2EE platform, the Java platform has become the standard for middle-tier development. This lets developers take advantage of the robustness, multithreading, and security features that the Java programming language offers plus features such as enhanced security, connection pooling, and distributed transactions available with the J2EE platform.

Thus, the JDBC API plays an essential role in both two-tier and three-tier architectures. With enterprises using the Java programming language for writing server code, the JDBC API is being used extensively in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets, all of which are explained later in the book. And, of course, the JDBC API is what allows access to a data source from a middle tier written in the Java programming language.

1.4.5   SQL Conformance

SQL is the standard language for accessing relational databases. Unfortunately, SQL is not yet as standard as one would like.

One area of difficulty is that data types used by different DBMSs (DataBase Management Systems) sometimes vary, and the variations can be significant. JDBC deals with this by defining a set of generic SQL type identifiers in the class `java.sql.Types`. Note that, as used in this book, the terms "JDBC SQL type," "JDBC type," and "SQL type" are interchangeable and refer to the generic SQL type identifiers defined in `java.sql.Types`. There is a more complete discussion of data type conformance in "Mapping SQL and Java Types," starting on page 1065. The section "JDBC Types Mapped to Database-specific SQL Types," on page 1093, shows vendor-specific data types.

Another area of difficulty with SQL conformance is that although most DBMSs use a standard form of SQL for basic functionality, they do not conform to the more recently defined standard SQL syntax or semantics for more advanced functionality. For example, not all databases support stored procedures or outer joins, and those that do are not always consistent with each other. Also, support for SQL99 features and data types varies greatly. It is hoped that the portion of SQL that is truly standard will expand to include more and more functionality. In the meantime, however, the JDBC API must support SQL as it is.

One way the JDBC API deals with this problem is to allow any query string to be passed through to an underlying DBMS driver. This means that an application is free to use as much SQL functionality as desired, but it runs the risk of receiving an error on some DBMSs. In fact, an application query may be something other than SQL, or it may be a specialized derivative of SQL designed for specific DBMSs (for document or image queries, for example).

A second way JDBC deals with problems of SQL conformance is to provide ODBC-style escape clauses, which are discussed in "SQL Escape Syntax in Statements," on page 958. The escape syntax provides a standard JDBC syntax for several of the more common areas of SQL divergence. For example, there are escapes for date literals and for stored procedure calls.

For complex applications, JDBC deals with SQL conformance in a third way. It provides descriptive information about the DBMS by means of the interface `DatabaseMetaData` so that applications can adapt to the requirements and capabilities of each DBMS. Typical end users need not worry about metadata, but experts may want to refer to Chapter 15, "DatabaseMetaData," starting on page 449.

Because the JDBC API is used as a base API for developing database access tools and other APIs, it also has to address the problem of conformance for anything built on it. A JDBC driver must support at least ANSI SQL92 Entry Level. (ANSI SQL92 refers to the standards adopted by the American National Standards Institute in 1992. Entry Level refers to a specific list of SQL capabilities.) Note, however, that although the JDBC 2.0 API includes support for SQL99 and SQLJ, JDBC drivers are not required to support them.

Given the wide acceptance of the JDBC API by database vendors, connectivity vendors, Internet service vendors, and application writers, it has become the standard for data access from the Java programming language.

1.4.6   Products Based on JDBC Technology

The JDBC API is a natural choice for developers using the Java platform because it offers easy database access for Java applications and applets.

JDBC technology has gathered significant momentum since its introduction, and many products based on JDBC technology have been developed. You can monitor the status of these products by consulting the JDBC web site for the latest information. It can be found at the following URL:

```
http://java.sun.com/products/jdbc
```

1.4.7   JDBC Product Framework

Sun Microsystems provides a framework of JDBC product components:

- the JDBC driver manager (included as part of the Java 2 Platform)

- the JDBC–ODBC bridge (included in the Solaris and Windows versions of the Java 2 Platform)

- The JDBC API Test Suite (available from the JDBC web site)

The JDBC `DriverManager` class has traditionally been the backbone of the JDBC architecture. It is quite small and simple; its primary function is to connect Java applications to the correct JDBC driver and then get out of the way. With the availability of the `javax.naming` and `javax.sql` packages, it is now also possible to use a `DataSource` object registered with a Java Naming and Directory Interface (JNDI) naming service to establish a connection with a data source. Both means

of getting a connection can be still used, but using a `DataSource` object is recommended whenever possible.

The JDBC–ODBC bridge driver allows ODBC drivers to be used as JDBC drivers. It was implemented as a way to get JDBC technology off the ground quickly, providing a way to access some of the data sources for which there were no JDBC drivers. Currently, however, there are a large number of JDBC drivers available, which greatly reduces the need for the JDBC–ODBC bridge driver.

Even though the JDBC–ODBC bridge driver has been updated to include some of the more advanced features of the JDBC API, it is not intended for use developing products. It is intended to be used only for prototyping or when no JDBC driver is available.

A further component in the framework is the JDBC API driver test suite, which is aimed at driver developers. It comes in two versions: the JDBC API Test Suite v1.2.1 and the JDBC API Test Suite v.1.3.1. These test suites cover J2EE compatibility and indicate whether a driver is compatible with other products that conform to the J2EE specification. More complete information, including download information, can be found in the section "JDBC Test Suite," on page 1116.

1.4.8   JDBC Driver Types

The JDBC drivers that we are aware of at this time generally fit into one of four categories:

1. *JDBC–ODBC bridge driver plus ODBC driver:* The Sun Microsystems bridge product provides JDBC access via ODBC drivers. Note that ODBC binary code, and in many cases database client code, must be loaded on each client machine that uses this driver. As a result, this kind of driver is most appropriate on a corporate network where client installations are not a major problem or for application server code written in Java in a three-tier architecture.

2. *Native-API partly Java driver:* This kind of driver converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, IBM DB2, or other DBMSs. Note that, like the bridge driver, this style of driver requires that some operating system-specific binary code be loaded on each client machine.

3. *JDBC-Net pure Java driver:* This driver translates JDBC calls into a DBMS-independent net protocol, which is then translated to a DBMS protocol by a server. This net server middleware is able to connect its pure Java clients to many different databases. The specific protocol used depends on the vendor.

In general, this is the most flexible JDBC alternative. It is likely that all vendors of this solution will provide products suitable for intranet use. In order for these products to support Internet access as well, they must handle the additional requirements for security, access through firewalls, and so forth, that the Web imposes.

4. *Native-protocol pure Java driver:* This kind of driver converts JDBC calls directly into the network protocol used by DBMSs. This allows a direct call from the client machine to the DBMS server and is an excellent solution for intranet access. Several that are now available include Oracle, Sybase, IBM DB2, Borland InterBase, and Microsoft SQL Server.

Figure 1.3 illustrates various types of driver implementations.

.

Figure 1.3: JDBC Driver Implementations
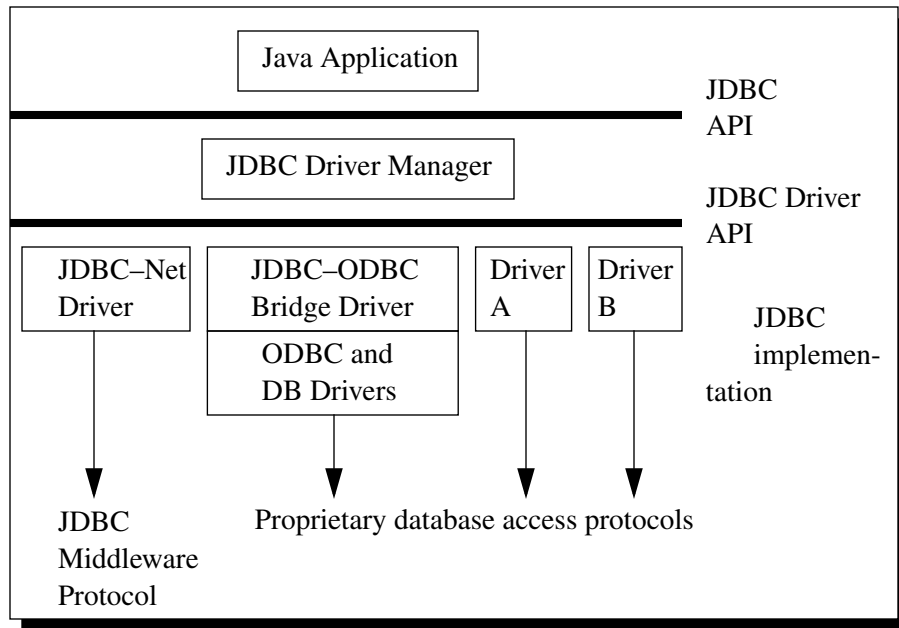
Driver categories 3 and 4 are the preferred way to access databases using the JDBC API. Driver categories 1 and 2 are interim solutions where direct pure Java drivers are not yet available. There are possible variations on categories 1 and 2 (not shown in the table "Driver Categories," on page 25) that require middleware,

but these are generally less desirable solutions. Categories 3 and 4 offer all the advantages of Java technology, including automatic installation (for example, downloading the JDBC driver with an applet that uses it).

Table 1.1 shows the four categories and their properties. The table uses the following definitions for types of network connections:

- Direct—a connection that a JDBC client makes directly to the DBMS server, which may be remote

- Indirect—a connection that a JDBC client makes to a middleware process that acts as a bridge to the DBMS server

Table 1.1: Driver Categories

Driver Category	All Java	Network Connection
1. JDBC–ODBC Bridge	No	Direct
2. Native API as basis	No	Direct
3. JDBC-Net	client, Yes server, Maybe	Indirect
4. Native protocol as basis	Yes	Direct

1.4.9   Obtaining JDBC Drivers

The web site for the JDBC API maintains a database with information about JDBC drivers, including what type they are and what functionality they support. There are currently over 200 drivers in this database, which you can search to find a driver that fits your needs. To get the latest information, check the web site at

```
http://industry.java.sun.com/products/jdbc/drivers
```

1.4.10   Java-relational DBMSs

A new generation of DBMSs that are Java-aware has been emerging. These new DBMSs, called Java-relational DBMSs, include new data types that allow an object in the Java programming language to be used as a column value in a database table. The JDBC 2.0 and 3.0 features support this new generation of DBMSs, and several database vendors are creating products with Java-relational capabilities. It should be

noted, however, that the 2.0 and 3.0 mechanisms are optional. If a DBMS does not support a particular feature, a JDBC driver is not required to implement it.

1.4.11 Other Products

Various application development tools using JDBC technology are under way. Watch the `java.sun.com/products/jdbc` web pages for updates.

1.5   The JDBC API and the Java Platforms

The JDBC API has become increasingly important to all three Java Platforms: the Java™ 2 Platform, Standard Edition (J2SE™); the Java™ 2 Platform, Enterprise Edition (J2EE™); and the Java™ 2 Platform, Micro Edition (J2ME™).

1.5.1   The JDBC API and the J2SE Platform

Database applications have long played an important role in business, scientific, government, and many other kinds of computer programs. The importance of the ability to store and retrieve data reliably cannot be overstated. In fact, database access is the foundation for the bulk of applications written today. As a result, JDBC is one of the essential APIs for J2SE, reflected by the fact that beginning with J2SE, version 1.4, the complete JDBC 3.0 API is bundled with the J2SE download. The core JDBC API has always been part of the J2SE platform, and now the JDBC Optional Package is also included.

1.5.2   The JDBC API and the J2EE Platform

Enterprise applications almost always depend on retrieving data from a DBMS. Furthermore, they often need to get data from more than one database server and handle an increasingly large volume of transactions. The `javax.sql` package provides the ability to pool database connections, thereby reducing the amount of resources needed and increasing performance. Added to that, it also provides the ability to use distributed transactions, which has become increasingly necessary, especially with the growth of Web services.

The J2EE platform simplifies the development of complex distributed applications by providing the "plumbing" for services such as security, distributed transactions, and connection pooling. A J2EE application server works with

JDBC `DataSource` implementations to supply the infrastructure required for distributed transactions and connection pooling.

The need for a J2EE application server to connect with a JDBC driver is so great that the JDBC team has been developing a component called the JDBC Connector. The JDBC Connector, based on the Connector 1.0 and Connector 1.5 specifications, allows any JDBC driver to be plugged in to any J2EE application server that adheres to the Connector specification requirements. As of this writing, the JDBC Connector is available as an early access release from the Java Developer Connection web site. Information is available at

```
http://java.sun.com/products/jdbc/related.html
```

The download includes documentation explaining how the JDBC Connector works and how to use it to plug a JDBC driver in to a J2EE application server. Driver vendors should check the section "Connectors," on page 1117.

1.5.3   The JDBC API and the J2ME Platform

The Java programming language is ideal for the burgeoning market of small devices. To make database access possible from the Java programming language for small devices, a community of experts is developing a subset of the JDBC API to achieve a smaller footprint while still providing useful database operations. These experts are developing this pared-down JDBC API through the Java Community Process[SM] as JSR (Java Specification Request) 169.

The remainder of this chapter gives a brief overview of the Java programming language and of SQL, a language for defining, accessing, and manipulating data in a relational database. Readers familiar with the Java programming language can skip "Java Overview," starting on page 27; readers familiar with SQL can skip "Relational Database Overview," starting on page 38.

1.6   Java Overview

The Java programming language is a powerful but lean object-oriented programming language. It originally generated a lot of excitement because it makes it possible to program for the Internet by creating applets, programs that can be embedded in a web page. The content of an applet is limited only by one's imagination. For example, an applet can be an animation with sound, an interactive game (that could

include various animations with sound), or a ticker tape with constantly updated stock prices. Applets can be just little decorations to liven up a web page, or they can be serious applications such as word processors or spreadsheets.

But Java technology is far more than a programming language for writing applets. It has become one of the standard languages for general-purpose and business programming. With the development of the Java 2 SDK, Enterprise Edition, and its "industrial strength" capabilities, Java technology has become more and more pervasive in the enterprise arena. There are many buzzwords associated with the Java platform, but because of its spectacular growth in popularity, one buzzword has taken hold: *ubiquitous*. Indeed, all indications are that it will soon be everywhere.

Java builds on the strengths of C++. It has taken the best features of C++ and discarded the more problematic and error-prone parts. To this lean core it has added garbage collection (automatic memory management), multithreading (the capacity for one program to do more than one thing at a time), and security capabilities. The result is that Java is simple, elegant, powerful, and easy to use.

Java is actually a platform consisting of three components: (1) the Java programming language, (2) the Java library of classes and interfaces, and (3) the Java Virtual Machine. The following sections will say more about these components.

1.6.1   Java Is Portable

One of the biggest advantages Java technology offers is that it is portable. An application written in the Java programming language will run on all of the major platforms. Any computer with a Java-based browser can run applets written in the Java programming language. A programmer no longer has to write one program to run on a Macintosh, another program to run on a Windows machine, still another to run on a Solaris or Linux machine, and so on. In other words, with Java technology, developers write their programs only once.

The Java Virtual Machine is what gives the Java programming language its cross-platform capabilities. Rather than being compiled into a machine language, which is different for each operating system and computer architecture, Java code is compiled into bytecodes. This makes Java applications bytecode portable.

With other languages, program code is compiled into a language that the computer can understand. The problem is that other computers with different machine instruction sets cannot understand that language. Java code, on the other hand, is compiled into bytecodes rather than a machine language. These bytecodes go to

the Java Virtual Machine, which executes them directly or translates them into the language that is understood by the machine running it.

In summary, with the JDBC API extending Java technology, a programmer writing Java code can access virtually any data source on any platform that supports the Java Virtual Machine.

1.6.2   Java Is Object-oriented

The Java programming language is object-oriented, which makes program design focus on *what* is being dealt with rather than on *how* to do something. This makes it more useful for programming in sophisticated projects because one can break things down into understandable components. A big benefit is that these components can then be reused.

Object-oriented languages use the paradigm of classes. In simplest terms, a class includes both data and the functions to operate on that data. You can create an *instance* of a class, also called an *object*, which will have all the data members and functionality of its class. Because of this, you can think of a class as being like a template, with each object being a specific instance of a particular type of class. For example, suppose you have a very simple class called Person, which has three *fields* (a data member is called a *field* in Java) and one *method* (a function is called a *method* in Java). The following code illustrates creating a simplified class. Don't worry if you don't understand everything in this example; just try to get a general idea. The first thing inside the beginning brace ({) is a constructor, a special kind of method that creates an instance of a class and sets its fields with their initial values.

```
public class Person {
    public Person(String n, int a, String oc) {
        name = n;
        age = a;
        occupation = oc;
    }
    public void identifySelf() {
        System.out.print("I am " + name + ", a " + age);
        System.out.println("-year-old " + occupation + ".");
    }
    protected String name;      // three attributes of Person
    protected int age;
```

```
    protected String occupation;
}
```

The last three items are *fields*, which are attributes of a Person object. They are given the access specifier protected, which means that these fields can be used by subclasses of Person but not by any other classes. (We will explain subclasses later in this section.) If the access specifier had been private, only the class Person could access these fields. The access specifier public allows access by all classes.

The following code creates an instance of Person and stores it in the variable *p*. This means that *p* is of type Person—a new class is a new type. This newly created instance of Person, *p*, is given the name, age, and occupation that were supplied to the constructor for Person. Note that the method new is used with a constructor to create a new instance of a class.

```
Person p = new Person("Adela", 37, "astronomer");
```

The following line of code causes *p* to identify herself. The results follow. (Note that the curved arrow signifies a line of output and is not part of the actual output.)

```
p.identifySelf();
```

➥ I am Adela, a 37-year-old astronomer.

The following code creates a second instance of Person and invokes the method identifySelf:

```
Person q = new Person("Hakim", 22, "student");
q.identifySelf();
```

➥ I am Hakim, a 22-year-old student.

The class paradigm allows one to *encapsulate* data so that specific data values or function implementations cannot be seen by those using the class. In the class Person, the fields name, age, and occupation are all given the access modifier protected, which signifies that these fields can be assigned values only by using methods in the class that defines them or in its subclasses—in this case, class Person or a subclass of Person.

In other words, the only way a user can change the value of private or protected fields in a class is to use the methods supplied by the class. In our example, the Person class does not provide any methods other than the constructor for assigning values, so in this case, a user cannot modify the name, age, or occupation of an instance of Person after its creation. Also, given the way Person is defined here, there is only one way a user can find out the values for name, age, and occupation: to invoke the method identifySelf.

To allow a user to modify, say, the field occupation, one could write a method such as the following:

```
public void setOccupation(String oc) {
    occupation = oc;
}
```

The following code defines methods that will return the current values of name and occupation, respectively:

```
public String getName() {
    return name;
}

public String getOccupation() {
    return occupation;
}
```

Using Person *p* in our example, the following code fragment sets the field occupation to a new value and verifies its current value.

```
p.setOccupation("orthodontist");
String newOccupation = p.getOccupation();
System.out.print(p.getName() + "'s new occupation is ");
System.out.println(newOccupation + ".");
```

➥ Adela's new occupation is orthodontist.

Encapsulation makes it possible to make changes in code without breaking other programs that use that code. If, for example, the implementation of a function is changed, the change is invisible to another programmer who invokes that function, and it doesn't affect his/her program, except, hopefully, to improve it.

The Java programming language includes *inheritance*, or the ability to derive new classes from existing classes. The *derived* class, also called a *subclass*, inherits all the data and functions of the existing class, referred to as the *parent* class or *superclass*. A subclass can add new data members to those inherited from the parent class. As far as methods are concerned, the subclass can reuse the inherited methods as they are, change them, and/or add its own new methods. For example, the subclass `VerbosePerson` could be derived from the class `Person`, with the difference between instances of the `Person` class and instances of the `VerbosePerson` class being the way they identify themselves. The following code creates the subclass `VerbosePerson` and changes only the implementation of the method `identifySelf`:

```java
public class VerbosePerson extends Person {
    public VerbosePerson(String n, int a, String oc) {
        super(n, a, oc);  // this calls the constructor for Person
    }

     // modifies the method identifySelf in class Person

    public void identifySelf() {
        System.out.println("Hi there! How are you doing today?");
        System.out.println("I go by the name of " + name + ".");
        System.out.print("I am " + age + " years old, and my ");
        System.out.println("occupation is " + occupation + ".");
    }
}
```

An instance of `VerbosePerson` will inherit the three protected data members that `Person` has, and it will have the method `identifySelf` but with a different implementation. The following code fragment creates an instance of the class `VerbosePerson`:

```java
VerbosePerson happyPerson = new VerbosePerson(
                            "Buster Brown", 45, "comedian");
```

A call to the method `identifySelf` will produce the following results:

```java
happyPerson.identifySelf();
```

➥ Hi there! How are you doing today?
➥ I go by the name of Buster Brown.
➥ I am 45 years old, and my occupation is comedian.

1.6.3   Java Makes It Easy to Write Correct Code

In addition to being portable and object-oriented, Java facilitates writing correct code. Programmers spend less time writing Java code and a lot less time debugging it. In fact, many developers have reported slashing development time by as much as two-thirds. The following is a list of some of the features that make it easier to write correct code in the Java programming language:

- **Garbage collection** automatically takes care of deallocating unused memory. If an object is no longer being used (has no references to it), then it is automatically removed from memory, or "garbage collected." Programmers don't have to keep track of what has been allocated and deallocated themselves, which makes their job a lot easier, but, more importantly, it stops memory leaks.

- **No pointers** eliminates a big source of errors. By using object references instead of memory pointers, problems with pointer arithmetic are eliminated, and problems with inadvertently accessing the wrong memory address are greatly reduced.

- **Strong typing** cuts down on run-time errors. Because of strong type checking, many errors are caught when code is compiled. Dynamic binding is possible and often very useful, but static binding with strict type checking is used when possible.

- **Exception handling** provides a safe mechanism for error recovery, which facilitates writing correct code.

- **Simplicity** makes the Java programming language easier to learn and use correctly. The Java programming language keeps it simple by having just one way to do something instead of having several alternatives, as in some languages. Java also stays lean by not including multiple inheritance, which eliminates the errors and ambiguity that arise when you create a subclass that inherits from two or more classes. To replace the capabilities that multiple inheritance provides, the Java programming language lets you add functionality to a class through the use of interfaces. See the next section for a brief explanation of interfaces.

1.6.4   Java Includes a Library of Classes and Interfaces

The Java platform includes an extensive class library so that programmers can use already-existing classes as is, create subclasses to modify existing classes, or implement interfaces to augment the capabilities of classes.

Both classes and interfaces contain data members (fields) and functions (methods), but there are major differences. In a class, fields may be either variable or constant, and methods are fully implemented. In an interface, fields must be constants, and methods are just prototypes with no implementations. The prototypes give the method signature (the return type, the function name, and the number of parameters with the type for each parameter), but the programmer must supply implementations. To use an interface, a programmer defines a class, declares that it implements the interface, and then implements all of the methods in that interface as part of the class.

These methods are implemented in a way that is appropriate for the class in which the methods are being used. For example, suppose a programmer has created a class `Person` and wants to use an interface called `Sortable`, which contains various methods for sorting objects. If the programmer wanted to be able to sort instances of the `Person` class, she would declare the class to implement `Sortable` and write an implementation for each method in the `Sortable` interface so that instances of `Person` would be sorted by the criteria the she supplies. For instance, `Person` objects could be sorted by age, by name, or by occupation, and the order could be ascending or descending. Interfaces let a programmer add functionality to a class and give a great deal of flexibility in doing it. In other words, interfaces provide most of the advantages of multiple inheritance without its disadvantages.

A *package* is a collection of related classes and interfaces. The following list, though not complete, gives examples of some Java packages and what they cover.

- `java.lang`—the basic classes. This package is so basic that it is automatically included in any Java program. It includes classes dealing with numerics, strings, objects, run time, security, and threads.

- `java.io`—classes that manage reading data from input streams and writing data to output streams

- `java.util`—miscellaneous utility classes, including generic data structures, bit sets, time, date, string manipulation, random number generation, system properties, notification, and enumeration of data structures

- `java.net`—classes for network support

- `java.swing`—a set of graphical user interface (GUI) components that allow for portable graphical applications between Java enabled platforms. GUI components include user interface components such as windows, dialog boxes, buttons, checkboxes, lists, menus, scrollbars, and text fields.

- `java.applet`—the `Applet` class, which provides the ability to write applets; this package also includes several interfaces that connect an applet to its document and to resources for playing audio

- `java.sql`—the JDBC core API, which has classes and interfaces for accessing data sources

- `java.beans`—classes for creating reusable components (known as JavaBeans™ components, or Beans), which are typically, though not necessarily, graphical user interface components

- `java.rmi`—the package that lets Java applications make remote method invocations on Java objects

- `java.security`—the security framework for applications written in the Java programming language

- `org.omg.CORBA`—the main package for Java IDL, which allows Java applications to call remote CORBA objects

- `javax.naming`—a unified interface to multiple naming and directory services in the enterprise

- `javax.sql`—the package that provides server-side database capabilities such as connection pooling and distributed transactions

1.6.5   Java Is Extensible

A big plus for the Java programming language is the fact that it can be extended. It was purposely written to be lean with the emphasis on doing what it does very well; instead of trying to do everything from the beginning, it was written so that extending it is easy. Programmers can modify existing classes or write their own new classes. They can also write whole new packages or expand existing ones. For example, the JDBC 2.0 API greatly expanded the `java.sql` package and added an entirely new package, the `javax.sql` package. The JDBC 3.0 API added two new interfaces and many new methods to existing interfaces.

In addition to extensions, there are also many tools being developed to make existing capabilities easier to use. For example, a variety of tools greatly simplify creating and laying out graphical user interfaces, such as menus, dialog boxes, buttons, and so on.

1.6.6   Java Is Secure

It is important that a programmer not be able to write subversive code for applications or applets. This is especially true with the Internet being used more and more extensively for Web services and the electronic distribution of software and multimedia content.

The Java platform builds in security in four ways:

- **The way memory is allocated and laid out.** In Java technology, an object's location in memory is not determined until run time, as opposed to C and C++, where the compiler makes memory layout decisions. As a result, a programmer cannot look at a class definition and figure out how it might be laid out in memory. Also, since the Java programming language has no pointers, a programmer cannot forge pointers to memory.

- **The way incoming code is checked.** The Java Virtual Machine does not trust any incoming code and subjects it to what is called bytecode verification. The bytecode verifier, part of the Virtual Machine, checks that (1) the format of incoming code is correct, (2) incoming code doesn't forge pointers, (3) it doesn't violate access restrictions, and (4) it accesses objects as what they are (for example, an `InputStream` object is used only as an `InputStream` object).

- **The way classes are loaded.** The Java bytecode loader, another part of the Virtual Machine, checks whether classes loaded during program execution are local or from across a network. Imported classes cannot be substituted for built-in classes, and built-in classes cannot accidentally reference classes brought in over a network.

- **The way access is restricted for untrusted code.** The Java security manager allows users to restrict untrusted Java applets so that they cannot access the local network, local files, and other resources.

1.6.7   Java Performs Well

Java's performance is better than one might expect. Java's many advantages, such as having built-in security and being interpreted as well as compiled, do have a cost attached to them. However, various optimizations have been built in, and the bytecode interpreter can run very fast because it does not have to do any checking. As a result, code written in the Java programming language has done quite respectably in performance tests. Its performance numbers for interpreted bytecodes are usually more than adequate to run interactive graphical end-user applications. For situations that require unusually high performance, bytecodes can be translated on the fly, generating the final machine code for the particular CPU on which the application is running, at run time.

High-level interpreted scripting languages generally offer great portability and fast prototyping but poor performance. Low-level compiled languages such as C and C++ offer great performance but require large amounts of time for writing and debugging code because of problems with areas such as memory management, pointers, and multiple inheritance. Java offers good performance with the advantages of high-level languages but without the disadvantages of C and C++. In the world of design trade-offs, and with its performance being continually upgraded, the Java programming language provides a very attractive alternative.

1.6.8   Java Scales Well

The Java platform is designed to scale well, from portable consumer electronic devices (PDAs) to powerful desktop and server machines. The Java Virtual Machine takes a small footprint, and Java bytecode is optimized to be small and compact. As a result, the Java platform accommodates the need for both low storage and low bandwidth transmission over the Internet.

1.6.9   Java Is Multithreaded

Multithreading is simply the ability of a program to do more than one thing at a time. For example, an application could be faxing a document at the same time it is printing another document. Or, a program could process new inventory figures while it maintains a feed of current prices. Multithreading is particularly important in multimedia, where a program might often be running a movie, running an audio track, and displaying text all at the same time.

1.7   Relational Database Overview

A database is a means of storing information in such a way that information can be retrieved from it. In simplest terms, a *relational* database is one that presents information in tables with rows and columns. A table is referred to as a *relation* in the sense that it is a collection of objects of the same type (rows). Data in a table can be related according to common keys or concepts, and the ability to retrieve related data from a table is the basis for the term *relational database*. A Database Management System (DBMS) handles the way data is stored, maintained, and retrieved. In the case of a relational database, a Relational Database Management System (RDBMS) performs these tasks. DBMS as used in this book is a general term that includes RDBMS.

1.7.1   Integrity Rules

Relational tables follow certain integrity rules to ensure that the data they contain stay accurate and are always accessible. First, the rows in a relational table should all be distinct. If there are duplicate rows, there can be problems resolving which of two possible selections is the correct one. For most DBMSs, the user can specify that duplicate rows are not allowed, and if that is done, the DBMS will prevent the addition of any rows that duplicate an existing row.

A second integrity rule of the traditional relational model is that column values must not be repeating groups or arrays. A third aspect of data integrity involves the concept of a *null* value. A database takes care of situations where data may not be available by using a null value to indicate that a value is missing. It does not equate to a blank or zero. A blank is considered equal to another blank, a zero is equal to another zero, but two null values are not considered equal.

When each row in a table is different, it is possible to use one or more columns to identify a particular row. This unique column or group of columns is called a *primary key*. Any column that is part of a primary key cannot be null; if it were, the primary key containing it would no longer be a complete identifier. This rule is referred to as *entity integrity*. (The rule for *referential integrity* is discussed in the section "Joins," on page 42.)

Table 1.2 illustrates some of these relational database concepts. It has five columns and six rows, with each row representing a different employee.

Table 1.2:   Employees

Employee_ Number	First_Name	Last_Name	Date_of_Birth	Car_ Number
10001	Axel	Washington	28-AUG-43	5
10083	Arvid	Sharma	24-NOV-54	null
10120	Jonas	Ginsburg	01-JAN-69	null
10005	Florence	Wojokowski	04-JUL-71	12
10099	Sean	Washington	21-SEP-66	null
10035	Elizabeth	Yamaguchi	24-DEC-59	null

The primary key for this table would generally be the employee number because each one is guaranteed to be different. (A number is also more efficient than a string for making comparisons.) It would also be possible to use First_Name and Last_Name because the combination of the two also identifies just one row in our sample database. Using the last name alone would not work because there are two employees with the last name of "Washington." In this particular case the first names are all different, so one could conceivably use that column as a primary key, but it is best to avoid using a column where duplicates could occur. If Elizabeth Taylor gets a job at this company and the primary key is First_Name, the RDBMS will not allow her name to be added (if it has been specified that no duplicates are permitted). Because there is already an Elizabeth in the table, adding a second one would make the primary key useless as a way of identifying just one row. Note that although using First_Name and Last_Name is a unique composite key for this example, it might not be unique in a larger database. Note also that Table 1.2 assumes that there can be only one car per employee.

1.7.2   SELECT Statements

SQL is a language designed to be used with relational databases. There is a set of basic SQL commands that is considered standard and is used by all RDBMSs. For example, all RDBMSs use the SELECT statement.

A SELECT statement, also called a *query*, is used to get information from a table. It specifies one or more column headings, one or more tables from which to select, and some criteria for selection. The RDBMS returns rows of the column

entries that satisfy the stated requirements. A SELECT statement such as the following will fetch the first and last names of employees who have company cars:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number IS NOT NULL
```

The result set (the set of rows that satisfy the requirement of not having null in the Car_Number column) follows. The first name and last name are printed for each row that satisfies the requirement because the SELECT statement (the first line) specifies the columns First_Name and Last_Name. The FROM clause (the second line) gives the table from which the columns will be selected.

```
FIRST_NAME            LAST_NAME
----------            -----------
Axel                  Washington
Florence              Wojokowski
```

The following code produces a result set that includes the whole table because it asks for all of the columns in the table Employees with no restrictions (no WHERE clause). Note that "SELECT *" means "SELECT all columns."

```
SELECT *
FROM Employees
```

1.7.3   WHERE Clauses

The WHERE clause in a SELECT statement provides the criteria for selecting values. For example, in the following code fragment, values will be selected only if they occur in a row in which the column Last_Name begins with the string 'Washington'.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE 'Washington%'
```

The keyword LIKE is used to compare strings, and it offers the feature that patterns containing wildcards can be used. For example, in the code fragment above, there is a percent sign (%) at the end of 'Washington', which signifies that

any value containing the string 'Washington' plus zero or more additional characters will satisfy this selection criterion. So 'Washington' or 'Washingtonian' would be matches, but 'Washing' would not be. The other wildcard used in LIKE clauses is an underbar (_), which stands for any one character. For example,

```
WHERE Last_Name LIKE 'Ba_man'
```

would match 'Batman', 'Barman', 'Badman', 'Balman', 'Bagman', 'Bamman', and so on.

The code fragment below has a WHERE clause that uses the equal sign (=) to compare numbers. It selects the first and last name of the employee who is assigned car 12.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number = 12
```

The next code fragment selects the first and last names of employees whose employee number is greater than 10005:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number > 10005
```

WHERE clauses can get rather elaborate, with multiple conditions and, in some DBMSs, nested conditions. This overview will not cover complicated WHERE clauses, but the following code fragment has a WHERE clause with two conditions; this query selects the first and last names of employees whose employee number is less than 10100 and who do not have a company car.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number < 10100 and Car_Number IS NULL
```

A special type of WHERE clause involves a join, which is explained in the next section.

1.7.4   Joins

A distinguishing feature of relational databases is that it is possible to get data from more than one table in what is called a *join*. Suppose that after retrieving the names of employees who have company cars, one wanted to find out who has which car, including the make, model, and year of car. This information is stored in another table, Cars, shown in Table 1.3.

Table 1.3: Cars

Car Number	Make	Model	Year
5	Honda	Civic DX	1996
12	Toyota	Corolla	1999

There must be one column that appears in both tables in order to relate them to each other. This column, which must be the primary key in one table, is called the *foreign key* in the other table. In this case, the column that appears in two tables is Car_Number, which is the primary key for the table Cars and the foreign key in the table Employees. If the 1996 Honda Civic were wrecked and deleted from the Cars table, then Car_Number 5 would also have to be removed from the Employees table in order to maintain what is called *referential integrity*. Otherwise, the foreign key column (Car_Number) in Employees would contain an entry that did not refer to anything in Cars. A foreign key must either be null or equal to an existing primary key value of the table to which it refers. This is different from a primary key, which may not be null. There are several null values in the Car_Number column in the table Employees because it is possible for an employee not to have a company car.

The following code asks for the first and last names of employees who have company cars and for the make, model, and year of those cars. Note that the FROM clause lists both Employees and Cars because the requested data is contained in both tables. Using the table name and a dot (.) before the column name indicates which table contains the column.

```
SELECT Employees.First_Name, Employees.Last_Name, Cars.Make,
         Cars.Model, Cars.Year
FROM Employees, Cars
WHERE Employees.Car_Number = Cars.Car_Number
```

This returns a result set that will look similar to the following:

```
FIRST_NAME    LAST_NAME     MAKE      MODEL      YEAR
-----------   ------------  --------  ---------  -------
Axel          Washington    Honda     CivicDX    1996
Florence      Wojokowski    Toyota    Corolla    1999
```

1.7.5   Common SQL Commands

SQL commands are divided into categories, the two main ones being Data Manipulation Language (DML) commands and Data Definition Language (DDL) commands. DML commands deal with data, either retrieving it or modifying it to keep it up-to-date. DDL commands create or change tables and other database objects such as views and indexes.

A list of the more common DML commands follows:

- SELECT—used to query and display data from a database. The SELECT statement specifies which columns to include in the result set. The vast majority of the SQL commands used in applications are SELECT statements.

- INSERT—adds new rows to a table. INSERT is used to populate a newly created table or to add a new row (or rows) to an already-existing table.

- DELETE—removes a specified row or set of rows from a table

- UPDATE—changes an existing value in a column or group of columns in a table

The more common DDL commands follow:

- CREATE TABLE—creates a table with the column names the user provides. The user also needs to specify a type for the data in each column. Data types vary from one RDBMS to another, so a user might need to use metadata to establish the data types used by a particular database. (See "Metadata," on page 46, for a definition of metadata. Also, the table "JDBC

Types Mapped to Database-specific SQL Types," on page 1093, shows the type names used by some leading DBMSs.) CREATE TABLE is normally used less often than the data manipulation commands because a table is created only once, whereas adding or deleting rows or changing individual values generally occurs more frequently.

- DROP TABLE—deletes all rows and removes the table definition from the database. A JDBC API implementation is required to support the DROP TA-BLE command as specified by SQL92, Transitional Level. However, support for the CASCADE and RESTRICT options of DROP TABLE is optional. In addition, the behavior of DROP TABLE is implementation-defined when there are views or integrity constraints defined that reference the table being dropped.

- ALTER TABLE—adds or removes a column from a table; also adds or drops table constraints and alters column attributes

1.7.6   Result Sets and Cursors

The rows that satisfy the conditions of a query are called the *result set*. The number of rows returned in a result set can be zero, one, or many. A user can access the data in a result set one row at a time, and a *cursor* provides the means to do that. A cursor can be thought of as a pointer into a file that contains the rows of the result set, and that pointer has the ability to keep track of which row is currently being accessed. A cursor allows a user to process each row of a result set from top to bottom and consequently may be used for iterative processing. Most DBMSs create a cursor automatically when a result set is generated.

The JDBC 2.0 API added new capabilities for a result set's cursor, allowing it to move both forward and backward and also allowing it to move to a specified row or to a row whose position is relative to another row.

1.7.7   Transactions

When one user is accessing data in a database, another user may be accessing the same data at the same time. If, for instance, the first user is updating some columns in a table at the same time the second user is selecting columns from that same table, it is possible for the second user to get partly old data and partly updated data. For this reason, DBMSs use *transactions* to maintain data in a consistent state (*data*

consistency) while allowing more than one user to access a database at the same time (*data concurrency*).

A transaction is a set of one or more SQL statements that make up a logical unit of work. A transaction ends with either a *commit* or a *rollback*, depending on whether there are any problems with data consistency or data concurrency. The `commit` statement makes permanent the changes resulting from the SQL statements in the transaction, and the `rollback` statement undoes all changes resulting from the SQL statements in the transaction.

A *lock* is a mechanism that prohibits two transactions from manipulating the same data at the same time. For example, a table lock prevents a table from being dropped if there is an uncommitted transaction on that table. In some DBMSs, a table lock also locks all of the rows in a table. A row lock prevents two transactions from modifying the same row, or it prevents one transaction from selecting a row while another transaction is still modifying it.

Chapter 11, "Connection," has more information about transactions. See especially the sections "Transactions," on page 392, and "Transaction Isolation Levels," on page 393.

1.7.8   Stored Procedures

A *stored procedure* is a group of SQL statements that can be called by name. In other words, it is executable code, a mini-program, that performs a particular task that can be invoked the same way one can call a function or method. Traditionally, stored procedures have been written in a DBMS-specific programming language. The latest generation of database products allows stored procedures to be written using the Java programming language and the JDBC API. Stored procedures written in the Java programming language are bytecode portable between DBMSs. Once a stored procedure is written, it can be used and reused because a DBMS that supports stored procedures will, as its name implies, store it in the database.

The following code is an example of how to create a very simple stored procedure using the Java programming language. Note that the stored procedure is just a static Java method that contains normal JDBC code. It accepts two input parameters and uses them to change an employee's car number.

Do not worry if you do not understand the example at this point; it is presented only to illustrate what a stored procedure looks like. You will learn how to write the code in this example in the tutorials that follow. Specifically, the sections "SQL Statements for Creating a Stored Procedure," "Calling a Stored Procedure

Using the JDBC API," and "Stored Procedures Using SQLJ and the JDBC API,"
all in Chapter 2, "Basic Tutorial," explain more about writing stored procedures.

```java
import java.sql.*;

public class UpdateCar {

    public static void UpdateCarNum(int carNo, int empNo)
                                            throws SQLException {
        Connection con = null;
        PreparedStatement pstmt = null;

        try {
          con = DriverManager.getConnection("jdbc:default:connection");

            pstmt = con.prepareStatement(
                        "UPDATE EMPLOYEES SET CAR_NUMBER = ? " +
                        "WHERE EMPLOYEE_NUMBER = ?");
            pstmt.setInt(1, carNo);
            pstmt.setInt(2, empNo);
            pstmt.executeUpdate();
        }
        finally {
            if (pstmt != null) pstmt.close();
        }
    }
}
```
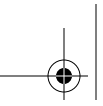
1.7.9   Metadata

Databases store user data, and they also store information about the database itself.
Most DBMSs have a set of system tables, which list tables in the database, column
names in each table, primary keys, foreign keys, stored procedures, and so forth.
Each DBMS has its own functions for getting information about table layouts and
database features. JDBC provides the interface DatabaseMetaData, which a driver
writer must implement so that its methods return information about the driver and/or
DBMS for which the driver is written. For example, a large number of methods

return whether or not the driver supports a particular functionality. This interface gives users and tools a standardized way to get metadata. See "`DatabaseMetaData` Overview," on page 449, and "`ResultSetMetaData` Overview," on page 783, for more information. In general, developers writing tools and drivers are the ones most likely to be concerned with metadata.