Java Database Connectivity (JDBC) API

JDBC is a set of Java interfaces and classes that connects our application to nearly any relational database management system. In this notes we will see how to connect with two different databases management system. Once connected we will see how to loop through rows of retrieve data and how to display their content. We will also see **prepareStatement** objects to create **parameterized SQL statements**.

I'll then show you how to insert, update, and delete data, and how to dynamically acquire information about your database structure. This notes will help you get started with this important Java API.

I have already installed Eclipse, and I'll go to the Eclipse Preferences dialog. I can get the Preferences either through Eclipse Preferences on Mac or from the Window menu on Windows. In the **Preferences** dialog, go to the **Java section** and then to the **Compiler** and make sure your Compiler compliance level is set to the latest version.

I'm going to reset the current Java perspective in the eclipse (**Opened section in side bars**). The Java perspective is installed as part of the Eclipse installation. I'll go **to window and Reset Perspective and click "Yes"**, and then I'm going to close the Task List view, which I won't need for this course. Then I'll save this reconfigure perspective as a custom perspective named "**Java DB**". I'll go to the menu and choose **Window > Save Prospective** As, and I'll name this Java DB and click OK. Now at any time during the exercises, I'll be able to return to this custom perspective, and this will be my starting perspective in each of the exercises of this course.

I'll need a local installation of **MySQL** and a copy of **phpMyAdmin**, a web-based management system that lets us manage a MySQL Database.

I can select from one of the many available software bundles which include Apache, MySQL, PHP, and phpMyAdmin. These include **WampServer** for Windows, **MAMP** for Mac OS X, or **XAMPP** or **BitNami**, which are both available for both Windows and Mac.

If you're using MAMP, there's one preferences thing you need to take care of. When you first install MAMP, you'll see that it's set ports of 8888 for Apache and 8889 for MySQL. It does this so you can run the copy of Apache included with the MAMP side by side with a copy of Apache included with the Mac.

Now the next step, once you have confirmed that MySQL is running is to import a database. I'll go to the Databases tab, I'll click into the text box, and I'll type the name of the database I want to create, "**explorecalifornia**". After that find **explorecalifornia.sql** in this current folder and run the all queries. If you want to refresh your database, you can just run this SQL script in the future, and it will delete all of the existing data and refresh the database table structure and content.

If you want to refresh your database, you can just run this SQL script in the future, and it will delete all of the existing data and refresh the database table structure and content. I'll be sure to select the file with the file extension **.sql**. There are also files with extensions of **.script** and .**properties**, but those are for use with another Database Management System.

What is JDBC?

I'm going to start by describing the nature and history of JDBC and how it fits into the larger Java World. JDBC is an API, a **set of interfaces and classes** that let you easily connect to relational databases, such as Oracle, SQL Server, MySQL, and many others. It was originally known as Java Database Connectivity, but more recently it's been known by simply the acronym JDBC.

JDBC was introduced into the Java programming language very early. It was included in JDK1.1 in **1997**, and has been a part of all releases of Java Standard Edition ever since.

The history of JDBC is a history of added features and improved performance. The original release in 1997 included the main classes and interfaces that you'll find yourself using all the time, including the Connection interface which lets you make your initial connection to a database, the Statement which encapsulate SQL code, and ResultSet which returns data from the server.

Later versions of JDBC improved the features of the API, including the ability to update your data without SQL, improved performance, pooled connections, scrolling, and more data types. More recent releases have included more data types, the ability to work with stored procedures, and the ability to get metadata, or lists of tables and column information, from your database. In the most recent versions of Java, JDBC 4.0 was released with Java 6, and it included the ability to load drivers automatically.

And in the most recent version of JDBC, JDBC 4.1 which is included in Java 7, there are features that let you reduce the amount of code it takes to work with your databases.

So who uses JDBC? It's most commonly used in web-based applications that are hosted in JEE or Java Enterprise Edition servers. These include JBoss, Tomcat, WebSphere, and others. Developers also commonly use JDBC when they're working on desktop applications or applets that are working either with local databases stored on the client computer or with remote databases accessed over the Internet.

Less common uses include JDBC in Android applications. Android has its own API for working with local databases, specifically **SQLite**, and so developers typically don't use JDBC there. And when you're working with larger databases making calls from Android applications, it's more common to make calls to those databases through web services hosted by **middleware** servers. But if you're a Java programmer, it is important to understand what JDBC is and how it works, because even if you're working through Android or through web services, someone somewhere is probably using JDBC somewhere in your calling chain, and it's useful to know how it works.

There are other ways of getting to databases without doing direct JDBC programming. There are **higher-level abstractions** that are delivered as part of the large application frameworks. For example, the Spring application framework includes something called JDBC Template. It simplifies the amount of code you have to write, but in the background it's using JDBC to talk to the database. There's a similar but perhaps less popular templating library called **RIFE**, which does basically the same thing. And then there are data mapping APIs, the most popular is Hibernate.

Hibernate is something called an object-relational mapping API. It represents your database structure with Java classes and objects. In the background it's still using JDBC to communicate with the database, but it simplifies the amount of code you have to write in your own application. And there are other mapping libraries, such as **iBATIS** from Apache, and the **Java Persistence API or JPA**, which is actually a part of the Java EE platform. Again, regardless of whether you use JDBC directly, or you use one of these higher-level application frameworks or data mapping APIs in the background, JDBC is at work.

After learning this topic, I’ll be able to gain a greater understanding of how the Java programming language connects to databases and what are some of the pitfalls and benefits of using JDBC.

Choosing JBDC Driver:

Applications that use the JDBC API **require drivers**. A JDBC driver is a software library that encapsulates the logic you need to communicate between your application and a database management system or a database file stored on the local hard disk. All JDBC drivers follow the same set of rules that is the API, that's defined in Java Standard Edition.

A driver library package will contain specific implementations of these Java interfaces. **Connection**, which lets you connect to the database, **ResultSet**, which encapsulates data returned from the database, **Statement**, **PreparedStatement**, and **CallableStatement**, which represent requests to the database, and many more.

Not all JDBC drivers will support all features of the JDBC spec. Check the documentation for your particular database and your particular driver to see what it's able to do. But nearly all JDBC drivers will support these five interfaces. Typically, we'll get our driver packages from the database vendors themselves, a MySQL driver from MySQL an Oracle driver for Oracle, and so on, but there are also third-party drivers available both free and commercial.

Type of Drivers:

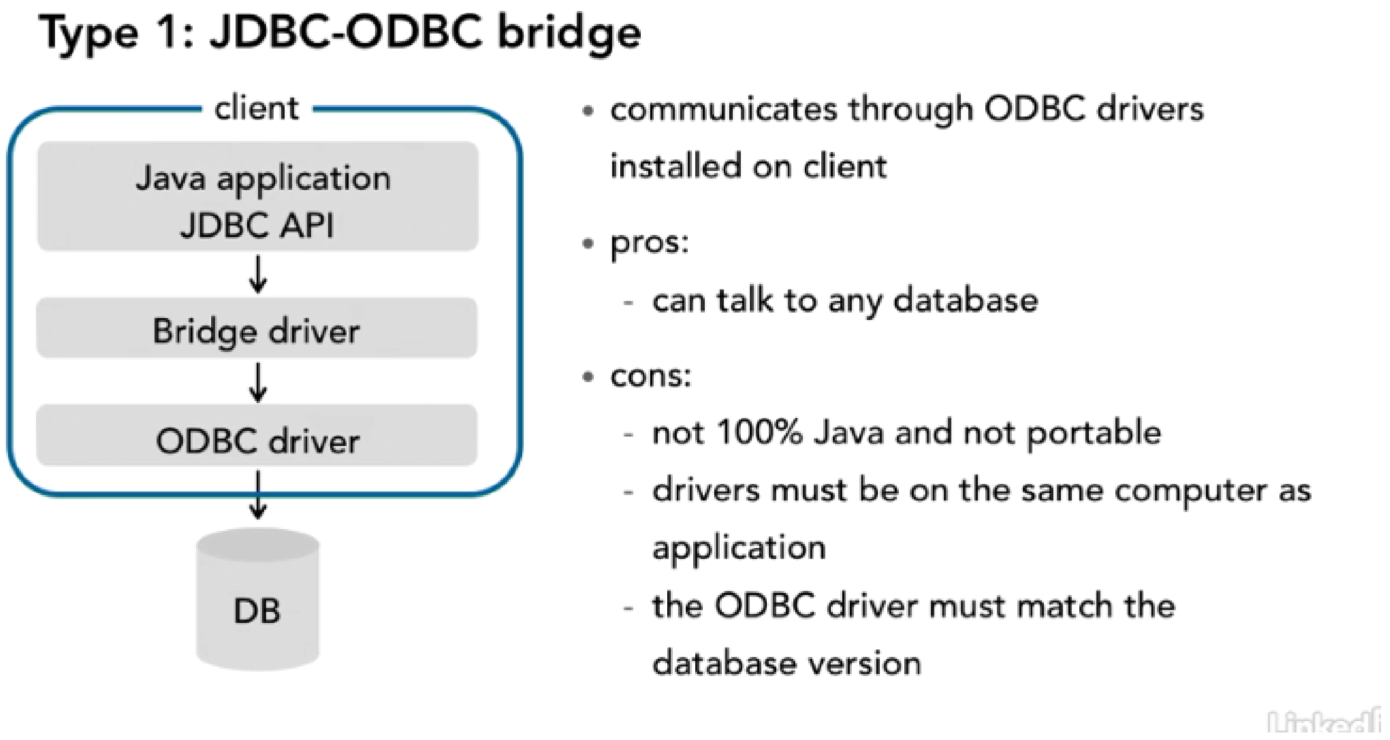
There are **four** distinct types of drivers, distinguished by their architecture.

The oldest type of driver is called the **Type 1** driver or the **JDBC-ODBC Bridge**. In the mid to late '90s when JDBC got started, ODBC or the **Open Database Connectivity protocol** was the dominant model for communicating with the database. And so Java first communicated with databases through ODBC. You would use the JDBC-ODBC Bridge driver, and then you would also use a separate ODBC driver that was designed for your specific database.

At runtime, requests would go from the application through the JDBC API to the Bridge driver from there to the ODBC driver and then to the database. This wasn't particularly fast, but it was dependable, and one of the pros was that you could talk to any database for which an ODBC driver existed, and that meant pretty much every relational database.

The downsides of the ODBC Bridge driver are that it's not 100% Java and therefore not portable between operating systems. Also you're working with two drivers not just one (**JDBC-ODBC Bridge driver and ODBC driver**), and they both have to be on the same computer as the application, so you have increased maintenance.

And finally, the ODBC driver has to match the database version, and so if you updated your database on the server, you'd have to go around to all the client applications and update those as well

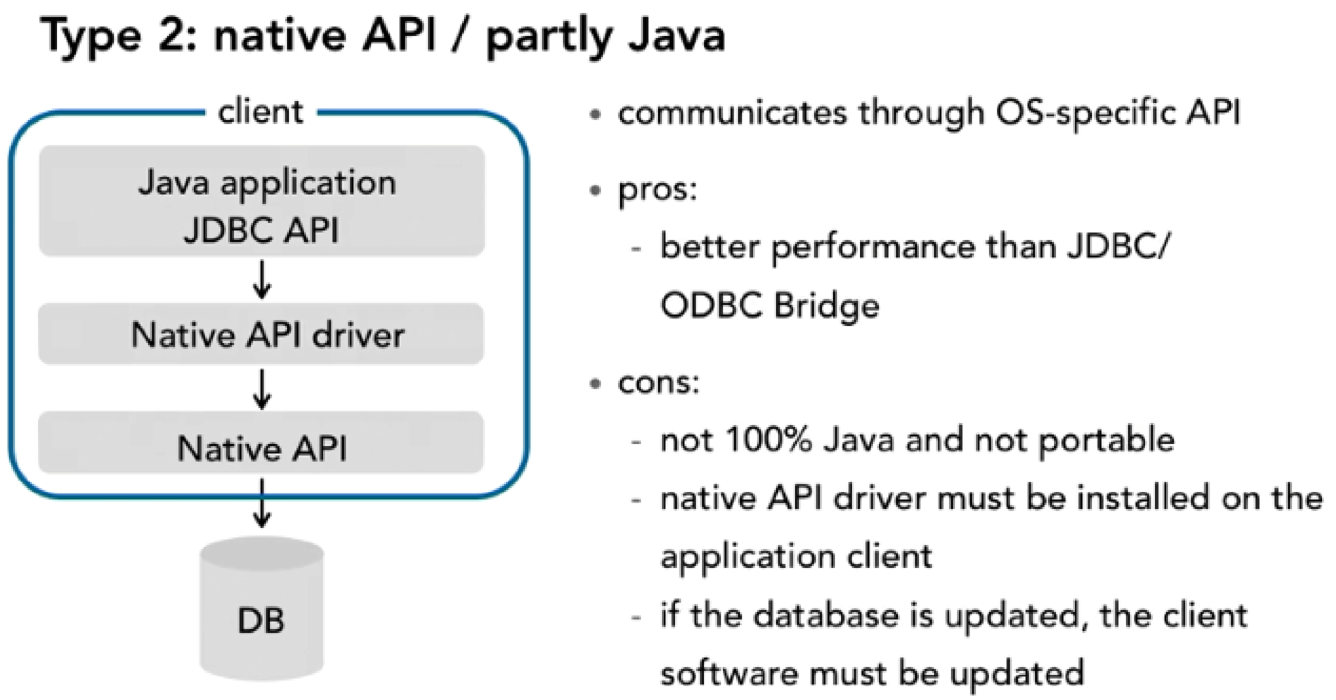


Type 1: JDBC-ODBC bridge

- communicates through ODBC drivers installed on client
- pros:
  - can talk to any database
- cons:
  - not 100% Java and not portable
  - drivers must be on the same computer as application
  - the ODBC driver must match the database version

The **Type 2 driver** combines a **native API** driver and a Java driver. They're both installed on the client system just like with the Bridge driver and the ODBC driver, but because you're working primarily with native APIs, the Type 2 driver tends to be very fast and can give you the best possible performance. Once again, just like the Bridge driver, you're not working in 100% Java, and therefore, your applications aren't portable between operating systems.

The native API driver has to be installed on the application client and maintained, and once again, if the database is updated, the client software has to be updated as well, but if you want the best possible performance, a Type 2 driver might be the way to go.

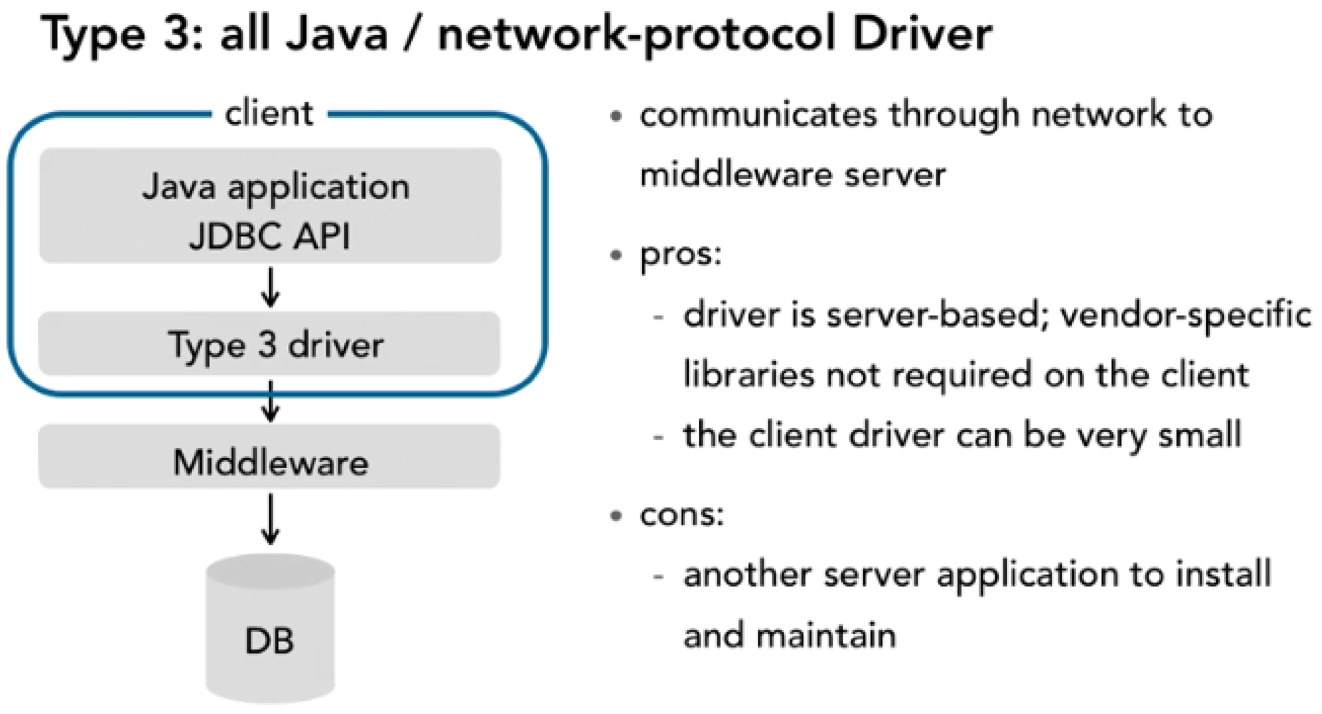


Type 2: native API / partly Java

- communicates through OS-specific API
- pros:
  - better performance than JDBC/ ODBC Bridge
- cons:
  - not 100% Java and not portable
  - native API driver must be installed on the application client
  - if the database is updated, the client software must be updated

The Type 3 driver actually is installed in multiple locations. You'll have 100% Java driver that's installed in the client along with the application, but then with the Type 3 driver architecture, you'll have a middleware server which hosts its own application, requests go at runtime from the application to the Type 3 driver that's installed on the client and from there over the network to the middleware server and from there to the database.

With a Type 3 driver, the middleware driver can be native, and so the communication between the middleware and the database can be very fast, but once again, you have maintenance challenges because you have more than one driver to maintain.



Type 3: all Java / network-protocol Driver

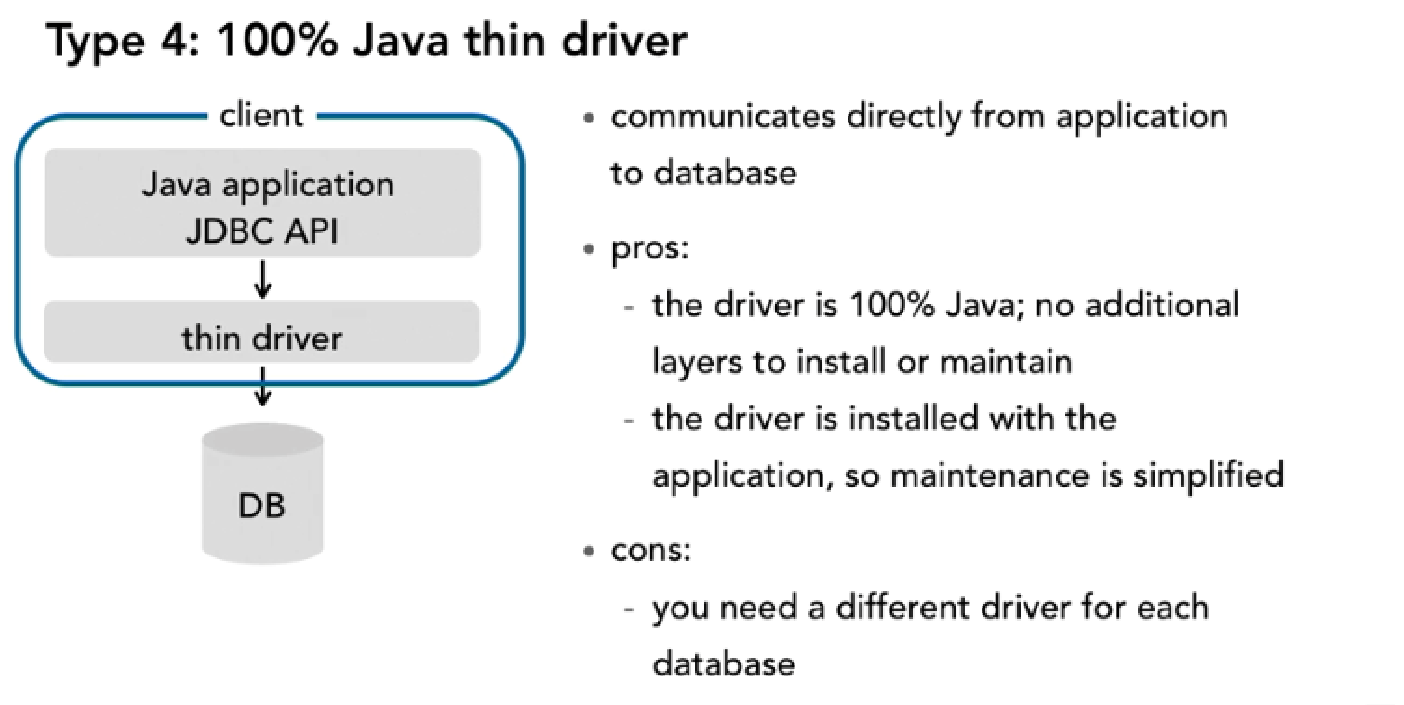
- communicates through network to middleware server
- pros:
  - driver is server-based; vendor-specific libraries not required on the client
  - the client driver can be very small
- cons:
  - another server application to install and maintain

The fourth type of driver, and one of the most common, is called the Type 4, or the **100% Java thin driver**. With a 100% Java driver, there is only one driver package, not two, and the driver is packaged with the Java application itself, whether on a client computer, or in a web environment on a Java Enterprise Edition application server.

Requests go from your application to the driver that's on the client and then your communications go from the application through JDBC through the thin driver to the database server if it's out on the web, or to the database file if it's on the local hard disk.

With the Java thin driver, we're communicating directly from the application to the database. There are no additional layers to install or maintain, and so maintenance is greatly simplified. The only downside is that you need a different driver package for each database you're working with.

Most applications will only use a single database type, but if you're working with more than one database management system, you'll need to provide multiple drivers. In this course I'll be doing all of my demonstrations with the two Type 4 drivers. One will be for the MySQL database server, and with this driver I'll show you how to communicate with a database server that's accessed over the web. I'll be actually working with a local copy of MySQL and addressing it as local host, but the code I'll be using would work just as well if the MySQL server were hosted out in the Cloud.



Type 4: 100% Java thin driver

- communicates directly from application to database
- pros:
  - the driver is 100% Java; no additional layers to install or maintain
  - the driver is installed with the application, so maintenance is simplified
- cons:
  - you need a different driver for each database

The other database I'll be working with in this course is called **HyperSQL**, or **HSQLDB**.

HyperSQL is one of many pure Java databases that are used by Java developers, others include the **Apache Derby database and H2**. These databases run as in-memory databases. But the database is initialized with a local file, but then it runs in the same Java process as the application.

So with these two drivers and these two database management systems, I'll be able to show you how to handle differences between databases while making your code as portable as possible, and I'll also be emphasizing the use of these Type 4 drivers, which are pure Java and can be encapsulated into your applications.

There are other types of drivers out there that aren't directly recognized by the JDBC API but can also be useful. For example, in 2010 there was talk of a Type 5 driver. A Type 5 driver has the same architecture as Type 4 that is it's 100% Java and Thin, but it gives you better performance than a pure Type 4 driver. And then there are other drivers that don't have specific type numbers that can run in the same Java process as the application just like the **HyperSQL** driver I'll show you. But if I stick with learning the four main types of drivers that are defined by JDBC, Types 1, 2, 3, and 4, those will cover the vast majority of JDBC drivers that are available to Java developers.

Connecting to MYSQL:

The first thing to learn in working with JDBC is how to connect to a database. I'm going to start with MySQL. To connect to MySQL, you'll need a **Java driver**, and you can get a free driver from **mysql.com/downloads**. On this screen, scroll down to the Connector section and look for **Connector/J**. You can get the most recent version of Connector/J from this website. MySQL Connector/J is the official JDBC driver for MySQL.