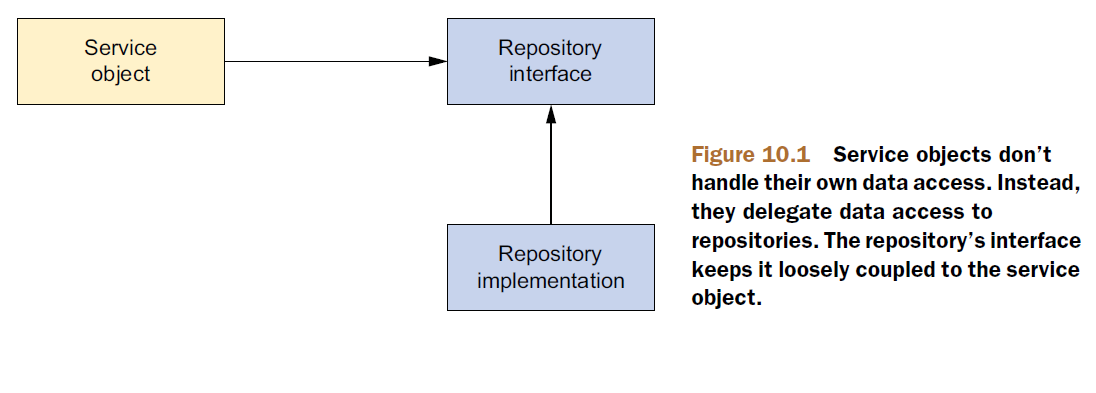
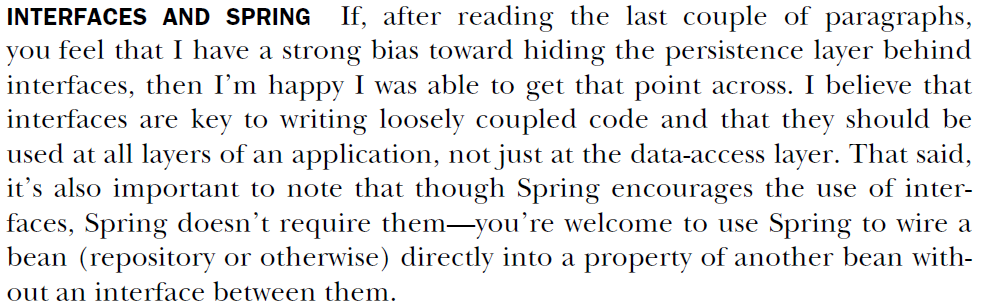
*Hitting the database with Spring and JDBC*

A perfect place to start is with a requirement of nearly any enterprise application: persisting data. You have probably dealt with database access in an application in the past. In practice, you’ll know that data access has many pitfalls. You have to initialize your data-access framework, open connections, handle various exceptions, and close connections. If you get any of this wrong, you could potentially corrupt or delete valuable company data.

* To avoid scattering persistence logic across all components in the application, it’s good to factor database access into one or more components that are focused on that task. Such components are commonly called data-access objects (DAOs) or repositories.
* To avoid coupling the application to any particular data-access strategy, properly written repositories should expose their functionality through interfaces.
* The service objects access the repositories through interfaces. This has a couple of positive consequences. First, it makes your service objects easily testable, because they’re not coupled to a specific data-access implementation. In fact, you could create mock implementations of these data-access interfaces. That would allow you to test your service object without ever having to connect to the database, which would significantly speed up your unit tests and rule out the chance of a test failure due to inconsistent data.
* In addition, the data-access tier is accessed in a persistence technology–agnostic manner. The chosen persistence approach is isolated to the repository, and only the relevant data-access methods are exposed through the interface. This makes for a flexible application design and allows the chosen persistence framework to be swapped out with minimal impact on the rest of the application.

**

* If the implementation details of the data-access tier were to leak into other parts of the application, the entire application would become coupled with the data-access tier, leading to a rigid application design.

**

***Getting to know Spring’s data-access exception hierarchy***

**JDBC’s SQLException.**

If you’ve ever written JDBC code (without Spring), you’re probably keenly aware that you can’t do anything with JDBC without being forced to catch SQLException. SQLException means something went wrong while trying to access a database. But there’s little about that exception that tells you what went wrong or how to deal with it.

* Some common problems that might cause a SQLException to be thrown include these:
* The application is unable to connect to the database.
* The query being performed has errors in its syntax.
* The tables and/or columns referred to in the query don’t exist.
* An attempt was made to insert or update values that violate a database constraint.
* The big question surrounding SQLException is how it should be handled when it’s caught. As it turns out, many of the problems that trigger a SQLException can’t be remedied in a catch block. Most SQLExceptions that are thrown indicate a fatal condition. If the application can’t connect to the database, that usually means the application will be unable to continue. Likewise, if there are errors in the query, little can be done about it at runtime.
* **If nothing can be done to recover from a** SQLException**, why are you forced to catch it?**
* Even if you have a plan for dealing with some SQLExceptions, you’ll have to catch the SQLException and dig around in its properties for more information about the nature of the problem. That’s because SQLException is treated as a one-size-fits-all exception for problems related to data access. Rather than have a different exception type for each possible problem, SQLException is the exception that’s thrown for all data-access problems.
* Some persistence frameworks offer a richer hierarchy of exceptions. Hibernate, for example, offers almost two dozen different exceptions, each targeting a specific data-access problem. This makes it possible to write catch blocks for the exceptions that you want to deal with.
* Even so, Hibernate’s exceptions are specific to Hibernate. As stated before, we’d like to isolate the specifics of the persistence mechanism to the data-access layer. If Hibernate-specific exceptions are being thrown, then the fact that you’re dealing with Hibernate will leak into the rest of the application. Either that, or you’ll be forced to catch persistence platform exceptions and rethrow them as platformagnostic

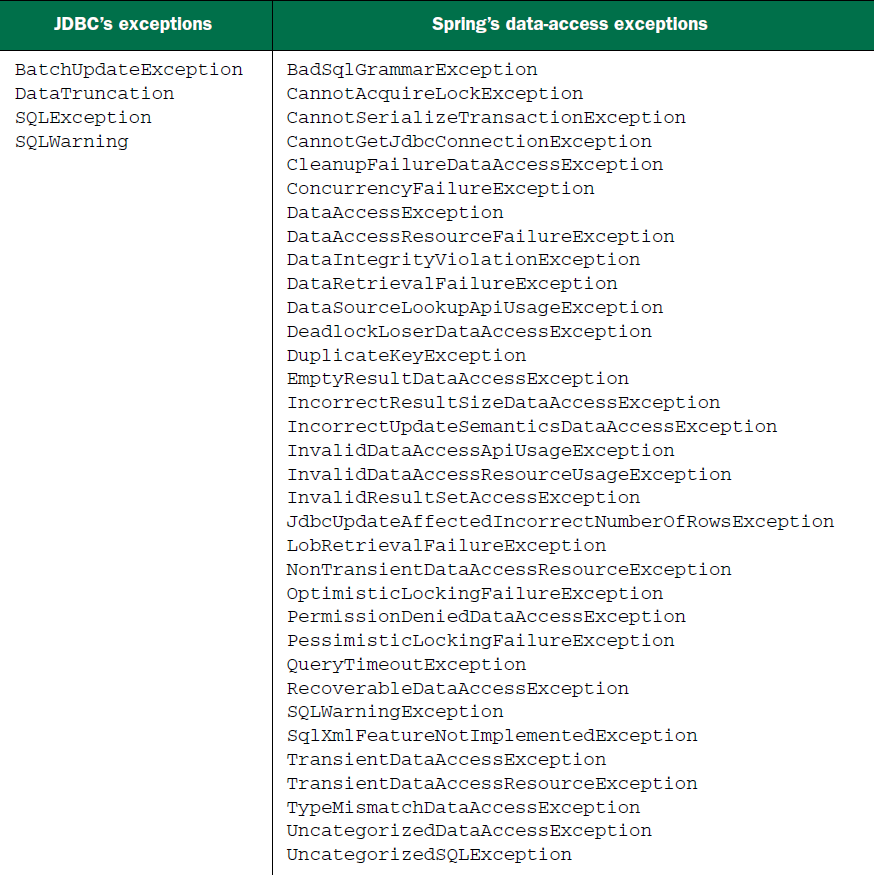
exceptions.

* On one hand, JDBC’s exception hierarchy is too generic—it’s not much of a hierarchy at all. On the other hand, Hibernate’s exception hierarchy is proprietary to Hibernate. What we need is a hierarchy of data-access exceptions that are descriptive but not directly associated with a specific persistence framework.

**SPRING’S PERSISTENCE PLATFORM–AGNOSTIC EXCEPTIONS**

Spring JDBC provides a hierarchy of data-access exceptions that solve both problems. In contrast to JDBC, Spring provides several data-access exceptions, each descriptive of the problem for which they’re thrown.

* Even though Spring’s exception hierarchy is far richer than JDBC’s simple SQLException,it isn’t associated with any particular persistence solution. This means you can count on Spring to throw a consistent set of exceptions, regardless of which persistence provider you choose. This helps to keep your persistence choice confined to the data-access layer.

****

* All of those exceptions are rooted with DataAccessException. What makes DataAccessException special is that it’s an unchecked exception. In other words, you don’t have to catch any of the data-access exceptions thrown from Spring (although you’re welcome to if you’d like).
* DataAccessException is just one example of Spring’s across-the-board philosophy of checked versus unchecked exceptions. Spring takes the stance that many exceptions are the result of problems that can’t be addressed in a catch block. Instead of forcing developers to write catch blocks (which are often left empty), Spring promotes the use of unchecked exceptions. This leaves the decision of whether or not to catch an exception in your hands.