5 State Management

To those searching for the truth—not the truth of dogma and darkness but the truth brought by reason, search, examination, and inquiry, discipline is required. For faith, as well intentioned as it may be, must be built on facts, not fiction—faith in fiction is a damnable false hope.

—Thomas Edison

A large part of the work of an enterprise system involves handling data. In fact, arguably, that's the only thing an enterprise system really does.

Although this was never really apparent during the era of the two-tier client/server system, the enterprise programmer needs to take care of two kinds of state: *transient state,* which is not yet a formal part of the enterprise data footprint, and *durable state,* which needs to be tracked regardless of what happens.

Transient state is data that the enterprise cares little about—in the event of a crash, nothing truly crucial has been lost, so no tears will be shed. The classic example of transient state is the e-commerce shopping cart. Granted, we don't ever want the system to crash, but let's put the objectivity glasses on for a moment: if the server crashes in the middle of a customer's shopping experience, losing the contents of the shopping cart, nothing is really lost (except for the time the customer spent building it up). Yes, the customer will be annoyed, but there are no implications to the business beyond that.

In a thick-client or rich-client application, transient state is pretty easy to handle: it's just the data stored in local variables in the client-side process that hasn't been preserved to the durable storage layer yet. Nothing particularly fancy needs be done to handle the lifecycle around it—when the client-side process shuts down, the transient state goes away with it.

In a thin-client, HTML-browser-based application, on the other hand, transient state takes on a whole new dimension. Because HTTP itself is a stateless protocol, with no intrinsic way to store per-client state, we've been forced to implement transient state mechanisms on top of the underlying plumbing. To most developers, this is exposed via the `HttpSession` mechanism that is part of the Servlet 2.x specifications. Unfortunately, nothing in life comes for free, and `HttpSession` definitely carries its share of costs, most notably to scalability of the entire system as a whole. Judicious use of per-client session state is crucial to a system that wants to scale to more than five concurrent users.

Durable state, on the other hand, is that which we normally think of when somebody starts to ask about "persistent data"—it's the data that has to be kept around for long periods of time. Officially, we'll define durable state as state that absolutely must be kept around even in the event of a JVM termination or crash, but since we could conceivably come up with situations where we'll want transient state to be stored in a database, it's more convenient to simply say that durable state is state that we care about.

Commonly, durable state has implicit legal and/or financial stakes—if, for example, you write a system that loses the items in a book order after the customer's credit card has been charged, you're exposing the company to a lawsuit, at the very least. Or, to flip it around, if you lose the fact that the customer hasn't been charged when the books ship, you're directly costing the company money and will likely find yourself filling out a new résumé pretty soon.

The distinction between the two is crucial when discussing state management because mechanisms that are useful for keeping transient state around won't necessarily be suitable for tracking durable state and vice versa. In some situations, we see some overlap, where we may want to track users' session state in a relational database in order to avoid any sort of situation where a user might lose that transient state. Perhaps it's not a commerce site at all but a human resources process that requires a long-running collection of forms to fill out, or a test that we don't want students to be able to "throw away" and start over just by closing the browser. In such situations, the distinction between transient and durable state may start to blur, but intuitively, it's usually pretty easy to spot the one from the other. Or, arguably, durable state *is* what we perceive at first to be transient state, as in the case of the students' test or the human resources

forms. Either way, drawing this distinction in your mind and designs will help keep straight which mechanisms should be used for state management in your systems.

Item 39: Use `HttpSession` sparingly

In order to maintain transient state on behalf of clients in an HTML/HTTP-based application, servlet containers provide a facility called *session space*, represented by the `HttpSession` interface. The idea itself is simple and straightforward: a servlet programmer can put any Serializable object (see Item 71) into session space, and the next time that same user issues a request to any part of the same Web application, the servlet container will ensure that the same objects will be in the `HttpSession` object when requested. This allows the servlet developer to maintain per-client state information on behalf of a Web application on the server across HTTP requests.

Unfortunately, this mechanism doesn't come entirely for free. In the first place, storing data on the server on behalf of every client reduces the resources available on that server, meaning the maximum load capability of the server goes down proportionally. It's a pretty simple equation: the more data stored into session space, the fewer sessions that machine can handle. So, it follows that in order to support as many clients as possible on a given machine, keep session storage to a minimum. In fact, for truly scalable systems, whenever possible, avoid using sessions whatsoever. By not incurring any per-client cost on the server, the machine load capacity (theoretically) goes to infinite, able to support however many clients can connect to it.

This suggestion to avoid sessions if possible goes beyond just simple scalability concerns. For servlet containers running within a Web farm, it's a necessity. Sessions are an in-memory construct; because memory is scoped to a particular machine, unless the Web farm has some mechanism by which a given client will always be sent back to the same server for every request, subsequent processing of a request will not find the session-stored objects placed there by an earlier request.

As it turns out, by the way, pinning HTTP requests against the same machine turns out to be frightfully hard to do. If the gateway tries to use

the remote address of the client as the indicator of the client request, it will run into issues on a couple of points. Internet service providers that supply IP addresses to dialup consumers, proxy servers, and NATs will offer the same IP address for multiple clients, thus accidentally putting all those clients against the same server. For a small number of clients behind the same proxy, this isn't a big deal, but if the proxy server in question is the one for AOL, this could be an issue.

The Servlet 2.2 Specification provides a potential solution to this session-within-clusters problem, in that if a servlet container supports it, a Web application can be marked with the `<distributable />` element in the deployment descriptor, and the servlet container will automatically ensure that session information is seamlessly moved between the nodes in the cluster as necessary, typically by serializing the session and shipping it over the network. (This is why the Servlet Specification requires objects placed in session to be Serializable.) On the surface, this seems to offer a solution to the problem, but it turns out to be harder than it first appears.

A possible mechanism to provide this support is to designate a single node in the cluster as a session state server, and on each request, whichever node is processing the request asks the session state server for the session state for the client, and this is shipped across the network to the processing node. This mechanism suffers from two side effects, however: (1) every request will incur an additional round-trip to the session state server, which increases the overall latency of the client request, but more importantly, (2) all session state is now being stored in a centralized server, which creates a single point of failure within the cluster. Avoiding single points of failure is frequently the reason we wanted to cluster in the first place, so this obviously isn't ideal.

A second possible mechanism is to take a more peer-to-peer approach. As a request comes into the node, the node issues a cluster-wide broadcast signal asking other nodes in the cluster whether they have the latest session state for this client. The node with the latest state responds, and the state is shipped to the processing node for use. This avoids the problem of a single point of failure, but we're still facing the additional round-trips to shift the session state around the network. Worse yet, as a client slowly makes the rounds through the cluster, a copy of that client's session state is stored on each node in the cluster, meaning now the cluster can support only the maximum number of clients storable on any single node in the cluster—this is obviously less than ideal. If each node in the cluster

throws away a client's session state after sending it to another node, however, it means that any possibility of caching the session state in order to avoid the network round-trip shipping the session state back and forth is lost.

The upshot of all this is that trying to build this functionality is not an easy task; few servlet containers have undertaken it. (Several of the EJB containers that are also servlet containers, such as WebLogic and WebSphere, support distributable Web applications, but this is typically done by building on top of whatever cluster support they have for stateful session beans. Needless to say, clustered stateful session bean state has the same issues.) Before trusting a servlet container to handle this, make sure to ask the vendors exactly *how* they do it, in order to understand the costs involved.

In the event that some kind of distributed session state mechanism is needed but the servlet container either doesn't provide it or provides a mechanism that is less than desirable for your particular needs, all is not lost. Thanks to the power of the Servlet 2.3 Specification, and filters, you can create your own distributable session mechanism without too much difficulty. The key lies in the fact that filters can nominate replacements for the `HttpServletRequest` and `HttpServletResponse` objects used within the servlet-processing pipeline.

The idea here is simple—create a filter that replaces the default `HttpServletRequest` with one that overrides the standard `getSession` method to return a customized `HttpSession` object instead of the standardized one. Logistically, it would look something like this:

```
import javax.servlet.*;
import javax.servlet.http.*;

public class DistributableSessionFilter
  implements Filter
{
  public void doFilter(ServletRequest request,
                       ServletResponse response,
                       FilterChain chain)
    throws ServletException
  {
    HttpServletRequest oldReq =
      (HttpServletRequest)request;
```

```
    HttpServletRequestWrapper newRequest =
      new HttpServletRequestWrapper(oldReq)
    {
      public HttpSession getSession(boolean create)
      {
        if (create)
          return new DistributedHttpSession(oldReq, response);
        else
        {
          // If user has created a distributed session already,
          // return it; otherwise return null
        }
      }
    };

    chain.doFilter(newRequest, response);
  }
}
```

In this code, `DistributedHttpSession` is a class that implements the `HttpSession` interface and whose `getAttribute`/`setAttribute` (and other) methods take the passed objects and store them to someplace "safe," such as the RDBMS or the shared session state server, and so on. Note that because the standard `HttpSession` object is no longer being used, it will be up to you, either in this filter or in the `DistributedHttpSession`, to set the session cookie in the HTTP response headers. Normally the servlet container itself handles this, but since we're not using its default session mechanism anymore, we have to pick up the slack. Use large random numbers for session identifiers, to prevent attackers from being able to guess session ID values, and make sure not to use the standard `JSESSIONID` header in order to avoid accidental conflicts with the servlet container itself.

The actual implementation of this fictitious `DistributedHttpSession` class can vary widely. One implementation is to simply store the session data into an RDBMS with hot replication turned on (to avoid a single-point-of-failure scenario); another is to use an external shared session state server; a third is to try the peer-to-peer approach. Whatever the implementation, however, the important thing is that by replacing the standard session system with your own version, you take control over

the exact behavior of the system, thus making this tunable across Web applications if necessary. This is a practical application of Item 6 at work.

Another session-related concern to be careful of is the *accidental* use of sessions. This isn't a problem within servlets written by hand by developers—a session isn't created until explicitly asked for via the call to `getSession` on the `HttpServletRequest`. However, this isn't the case with JSP pages. The JSP documentation states, quite clearly, that the directive to "turn on" session for a given JSP page (the `@page` directive with the `session` attribute) is set to `true` by default, meaning that the following JSP page has a session established for it, even though `session` is never used on the page:

```
<%@ page import="java.util.*">

<html>
<body>
Hello, world, I'm a stateless JSP page.
It is now <%= new Date() %>.
</body>
</html>
```

Worse yet, it takes only one JSP page anywhere in the Web application to do this, and the session (and the commensurate overhead associated with it, even if no objects are ever stored in it) will last for the entire time the client uses the Web application. As a result, make sure your JSPs all have session turned off by default, unless it's your desire to use sessions. I wish there were some way to establish `session = false` as the default for a given Web application, but thus far it's not the case.

Lest you believe otherwise, I'm not advocating the complete removal of all session use from your application; in fact, such a suggestion would be ludicrous, given that HTTP offers no reasonable mechanism for providing per-user state. When used carefully, `HttpSession` provides a necessary and powerful mechanism for providing per-user state within a Web application, and this is often a critical and necessary thing. (How else could I ensure that the user has successfully authenticated or track his or her progress through the application?) The danger is in overusing and abusing the mechanism, thereby creating additional strain on the servlet container when it's not necessary. Don't use sessions unless you need to, and when you do use them, make them lean and mean in order to keep the resources consumed on the servlet container machine as light as possible.

Item 40: Use objects-first persistence to preserve your domain model

You've spent weeks, if not months, designing an elegant object model representing your application's business domain objects, searching for just the right combination of inheritance, composition, aggregation, and other object-modeling techniques to build it. It has taken quite a while, but you're finally there—you have an object model the team can be proud of. Now it's time to preserve that domain data into the database, and you want to keep your object model in place; after all, what good is an object model if you're just going to have to turn around and write a whole bunch of ugly SQL to push the data back out to the database?

In an objects-first approach to persistence, we seek to keep the view of objects during persistence; this means either the objects know how to silently persist themselves without any prompting from us or they provide some kind of object-centric API for doing the persistence and retrieval. Thus, in the ideal world, writing code like the following would automatically create an entry in the database containing the person data for Stu Halloway, age 25:

```
Person p = new Person("Stu", "Halloway", 25);
System.out.println(p);
    // Prints "Stu Halloway, age 25"
```

Writing these next lines of code would automatically update the existing row created in the first snippet to bump Stu's age from 25 to 30:

```
Person p = Person.find("Stu", "Halloway");
System.out.println(p);
    // Prints "Stu Halloway, age 25"
p.setAge(30);
System.out.println(p);
    // Prints "Stu Halloway, age 30"
```

Notice one of the great benefits of the objects-first persistence approach: no ugly SQL, no worries about whether we need to do an `INSERT` or an `UPDATE`, and so on. All we see are objects, and we like it that way.

An objects-first approach tends to break down fairly quickly when trying to retrieve objects, however. Generally speaking, an objects-first approach takes one of two possible methods: either we issue queries for objects by creating objects that contain our criteria in pure object-oriented fashion or we use some kind of "query language" specific to object queries.

In a strictly purist objects-first environment, we never want to see anything but objects, so we end up building Query Objects [Fowler, 316[1]] that contain the criteria we're interested in restricting the query around. Unfortunately, building a complex query that executes by using criteria other than the object's primary key (sometimes called an *OID*, short for *object identifier*, in the parlance of OODBMSs is often complicated and/or awkward:

```
QueryObject q = new QueryObject(Person.class);
q.add(Criteria.and(
          Criteria.greaterThan("dependents", 2)),
              Criteria.lessThan("income", 80000)));
q.add(Criteria.and(
          Criteria.greaterThan("dependents", 0)),
              Criteria.lessThan("income", 60000)));
```

Here, we're trying to build the equivalent of the following lines:

```
SELECT * FROM person p
  WHERE ( (p.dependents > 2 AND p.income < 80000)
          OR (p.dependents > 0 AND p.income < 60000) )
```

Which is easier to read? Things get exponentially worse if we start doing deeply nested Boolean logic in the query, such as looking for "people making less than $80,000 with more than 2 dependents who in turn claim them as parents, or people making less than $60,000 with any dependents who in turn claim them as parents." In fact, it's not uncommon to find that an objects-first purist query approach has much stricter restrictions on what can be queried than a general-purpose query language, like SQL.

Which leads us to the second approach, that of creating some kind of "query language" for more concisely expressing queries without having to resort to overly complex code. All of the objects-first technology approaches in Java have ultimately fallen back to this: EJB 2.0 introduced EJBQL, a query language for writing finders for entity beans; JDO introduced JDOQL, which does the same for JDO-enhanced persistent classes; and going way back, OODBMSs used OQL, the Object Query Language. These languages are subtly different from each other, yet all share one defining similarity: they all look a lot like SQL, which is what we were

1. By the way, if you use Fowler's implementation of the Query Object found in his book, note that as written, the code could be vulnerable to a SQL injection attack (see Item 61).

trying to get away from in the first place. (JDOQL is technically a language solely for describing a filter, which is essentially just the predicate part of the query—the `WHERE` clause—while still using a Query Object–style API.) Worse, EJBQL as defined in EJB 2.0 lacks many of the key features of SQL that make it so powerful to use for executing queries. The 2.1 release will address some of this lack, but several features of SQL are still missing from EJBQL.

Another unfortunate side effect of using an objects-first approach is that of invisible round-trips; for example, when using the entity bean below, how many trips to the database are made?

```
PersonHome ph =
  (PersonHome)ctx.lookup("java:comp/env/PersonHome");

// Start counting round-trips from here
//
Collection personCollection = ph.findByLastName("Halloway");
for (Iterator i = personCollection.iterator(); i.hasNext(); )
{
  Person p = (Person)i.hasNext();
  System.out.println("Found " + p.getFirstName() +
                           " " + p.getLastName());
}
```

Although it might seem like just one round-trip to the database (to retrieve each person whose last name is Halloway and to populate the entity beans from the `PersonBean` pool as necessary), in fact, this is what's called the *N*+1 query problem in EJB literature—the finder call will look only for the primary keys of the rows matching the query criteria, populate the `Collection` with entity bean stubs that know only the primary key, and lazy-load the data into the entity bean as necessary. Because we immediately turn around and access data on the entity bean, this in turn forces the entity bean to update itself from the database, and since we iterate over each of the items in the `Collection`, we make one trip for the primary keys plus *N* more trips, where *N* equals the number of items in the collection.

Astute developers will quickly point out that a particular EJB entity bean implementation isn't necessarily required to do something like this; for example, it would be possible (if perhaps nonportable—see Item 11) to build an entity bean implementation that, instead of simply pulling back

the OIDs/primary keys of the entity as the result of a query, pulls back the entire data set stored within that entity, essentially choosing an eager-loading implementation rather than the more commonly used lazy-loading approach (see Items 47 and 46, respectively). Unfortunately, this would create a problem in the reverse—now we will complain about too much data being pulled across, rather than too little.

The crux of the problem here is that in an objects-first persistence scenario, the atom of data retrieval is the object itself—to pull back something smaller than an object doesn't make sense from an object-oriented perspective, just as it doesn't make sense to pull back something smaller than a row in an SQL query. So when all we want is the `Person`'s first name and last name, we're forced to retrieve the entire `Person` object to get it. Readers familiar with OQL will stand up and protest here, stating (correctly) that OQL, among others, allows for retrieval of "parts" of an object—but this leads to further problems. What exactly is the returned type of such a query? I can write something like the following lines:

```
SELECT p.FirstName, p.LastName
  FROM Person p
  WHERE p.LastName = 'Halloway';
```

But what, exactly, is returned? Normally, the return from an object query is an object of defined type (in the above case, a `Person` instance); what are we getting back here? There is no commonly accepted way to return just "part" of an object, so typically the result is something more akin to a `ResultSet` or Java `Map` (or rather, a `List` of `Map` instances).

Even if we sort out these issues, objects-first queries have another problem buried within them: object-to-object references. In this case, the difficulty occurs not so much because we can't come up with good modeling techniques for managing one-to-many, many-to-many, or many-to-one relationships within a relational database (which isn't trivial, by the way), but because when an object is retrieved, the question arises whether it should pull back all of its associated objects, as well. And how do we resolve the situation where two independent queries pull back two identical objects through this indirect reference?

For example, consider the scenario where we have four `Person` objects in the system: Stu Halloway is married to Joanna Halloway, and they have two children, Hattie Halloway and Harper Halloway. From any good object perspective, this means that a good `Person` model will have a field

for `spouse`, of type `Person` (or, more accurately, references to `Person`), as well as a field that is a collection of some type, called `children`, containing references to `Person` again.

So now, when we execute the earlier query and retrieve the first object (let's use Stu), should pulling the `Stu` object across the wire mean pulling `Joanna`, `Hattie`, and `Harper` across, as well? Again, should we eager-load the data—remember, these objects are referenced from fields in the `Stu` object instance—or lazy-load it? And when we move to the next result in the query, `Joanna`, which in turn references `Stu`, will we have one `Stu` object in the client process space or two? What happens if we do two separate queries, one to pull back `Stu` alone, and the second to pull back `Joanna`? This notion of object identity is important because in Java, object identity is established by the `this` pointer (the object's location), and in the database it's conveyed via the primary key—getting the two of them to match up is a difficult prospect, particularly when we throw transactions into the midst. It's not an impossible problem—an Identity Map [Fowler, 195] is the typical solution—but as an object programmer, you need to be aware of this problem in case your objects-first persistence mechanism doesn't take care of it.

The ultimate upshot here is that if you're choosing to take an objects-first persistence approach, there are consequences beyond "it's easier to work with." In many cases, the power and attractiveness of working solely with objects is enough to offset the pain, and that's saying a lot.

Item 41: Use relational-first persistence to expose the power of the relational model

For years (if not decades), the undisputed king of the data storage tier has been the relational database; despite a valiant run by OODBMSs in the mid-1990s, the relational database remains in firm control of enterprise data, and relational vendors appear unwilling to part with that position any time soon. We developers want to continue to use object-oriented languages, since we've found them to be most convenient for us to solve our problems, but businesses want to continue their investments in relational databases. So it seems natural that we would want to "hide" the messy details of persisting object data to the relational database. Unfortunately, therein lies a problem: objects and relations don't mix well. The

difficulty in achieving a good mapping between these two technologies even has its own name, the *object-relational impedance mismatch*.

Part of the problem of working with relational access technologies (like JDBC) from an object-oriented language is simply that the two technology bases view the world in very different ways. An object-oriented language wants to work with *objects*, which have *attributes* (fields) and *behaviors* (methods), whereas a relational technology sees the world as *tuples*, collections of data items grouped into a logical "thing"—what Date referred to as "relations" [Date]. In essence, a relational model deals with collections of relations, which we commonly refer to as *tables*; each relation is a *row*, each item in the tuple is a *column*, and an entire language built around the idea of manipulating data in this relational format provides access.

While a full treatise on the problems inherent in an object-relational mapping layer is well beyond the scope of this book, a brief look at just one of the problems involved may help explain why object-relational mapping is such a common problem in J2EE systems. Consider, for a moment, a simple domain object model:

```
public class Person
{
  private String firstName;
  private String lastName;
  private int age;

  // . . .
}

public class Employee extends Person
{
  private long employeeID;
  private float monthlySalary;
}
```

This is probably the world's simplest domain model, but how should we persist this out to a relational database?

One approach is to create two tables, `PERSON` and `EMPLOYEE`, and use a foreign-key relationship to tie rows from one to the other. This requires a join between these two tables every time we want to work with a given

`Employee`, which requires greater work on the part of the database on every query and modification to the data. We could store both `Person` and `Employee` data into a single `EMPLOYEE` table, but then when we create `Student` (extending `Person`) and want to find all `Person` objects whose last name is Smith, we'll have to search both `STUDENT` and `EMPLOYEE` tables, neither of which at a relational level have anything to do with one another. And if this inheritance layer gets any deeper, we're just compounding the problem even further, almost exponentially.

As if this weren't enough, more frequently than not, the enterprise developer doesn't have control over the database schema—it's already in use, either by legacy systems or other J2EE systems, or the schema has been laid down by developers in other groups. So even if we wanted to build a table structure to elegantly match the object model we built, we can't arbitrarily change the schema definitions.

From an entirely different angle, there may be other, far more practical reasons to abandon an objects-first approach. Perhaps no object-relational mapping product on the planet can deal with the relational database schema you inherited as part of your project, or you're simply more comfortable working with the relational model and SQL than an object model (although that would likely imply you started life as a database administrator and later became a Java programmer).

For these reasons and more, frequently it's easier to take a relational view of your data and embrace that fully by not hiding the relational access behind some other kind of encapsulatory wall, be it object-, procedural-, or hierarchical-oriented.

To understand what I mean by the idea of taking a relational-first approach, we need to take a step back for a moment and revisit what exactly the relational approach itself is. According to Chris Date, who along with E. F. Codd is considered to be one of the great fathers of the relational model, "relational systems are based on a formal foundation, or theory, called *the relational model of data*" [Date, 38, emphasis added]. For mathematicians, a relational model is based on set theory and predicate logic; for the rest of us, however, in simple terms, the relational model of data is seen as nothing but tables. Accessing data yields nothing but tables, and the operators (SQL) for manipulating that data produce tables from tables.

While this may seem like a pretty redundant discussion—after all, it takes about thirty seconds of looking at a relational database to figure out that

it's all about the tables—it's the "relation" in the relational model that provides much of the power. Because the end product of a relational data access (SQL statement) is a table, which is in turn the source of a relational data operator such as `JOIN`, `SELECT`, and so on, relational data access achieves what Date calls *closure*: results of one access can serve as the input to another. This gives us the ability to write nested expressions: expressions in which the operands themselves are represented by general expressions, instead of just by table names. This is where much of the power of SQL comes from, although we tend not to use it that much (typically because we keep trying to go for objects-first persistence approaches, and SQL nested expressions don't fit so well in an object-relational mapping layer).

Why is this such a big deal? Because SQL is a powerful language for accessing data out of a relational database, and thanks to the fact that everything produced by a SQL query is a table, we can have a single API for extracting however much, or however little, data from any particular query we need. We don't have the "smaller than an object" problem raised in Item 40 because everything comes back as a table, even if it's a table one column wide. We do face the problem that the relational model frequently won't match the object model programmers would rather work with, but we can address that in a bit. Let's first look at making the relational access itself easier.

Before you shrink in horror at the thought of being condemned to low-level JDBC access for the rest of your life, take a deep breath—a relational-first approach doesn't mean abandoning anything at a higher level than JDBC. Far from it, in fact. We can use several mechanisms to make relational access much easier from Java than just raw JDBC (which still remains as an option in many cases, despite its relatively low-level nature).

First, there's more to JDBC than just `Connection`, `Statement`, and `ResultSet` objects. `RowSet`, and in particular Sun's `CachedRowSet` implementation, frequently makes it much easier to work with JDBC by encapsulating the act of issuing the query and harvesting the results. So, assuming you're without a JDBC `DataSource`, issuing a query can be as easy as this:

```
RowSet rs = new WebRowSet();
    // Or use another RowSet implementation
```

```
// Provide RowSet with enough information to obtain a
// Connection
rs.setUrl("jdbc:dburl://dbserver/PEOPLE");
rs.setUsername("user");
rs.setPassword("password");

rs.setCommand("SELECT first_name, last_name FROM person " +
              "WHERE last_name=?");
rs.setString(1, "Halloway");

rs.execute();

// rs now holds the results of the query
```

Most of the calls to the `RowSet` could be hidden behind an object factory interface (see Item 72 for details on object factories), so that client code could be reduced to the following:

```
RowSet rs = MyRowSetFactory.getRowSet();
rs.setCommand(. . .);
rs.setString(1, "Halloway");
rs.execute();
```

If you ask me, that's about as simple as you're going to get for direct SQL-based access. The factory itself is not difficult to imagine:

```
public class MyRowSetFactory
{
  public static getRowSet()
  {
    RowSet rs = new WebRowSet(); // Or some other
                                 // implementation

    // This time, use JNDI DataSource
    // rather than url/username/password
    rs.setDataSourceName("java:comp/env/jdbc/PEOPLE_DS");

    return rs;
  }
}
```

The `RowSet` API has methods to control the details of how the query will be executed, including the fetch size of the `ResultSet` (which should be set as high as possible, preferably to the size of the rows returned by the query, to avoid round-trips at the expense of one giant "pull" across the network), and the transaction isolation level (see Item 35) of this call.

However, the `RowSet` API still requires that you author the SQL query you're interested in every time you want to use it, and it can get awfully tiring to type `SELECT blah blah blah FROM table WHERE baz=?` every time you want to pull some data back, particularly if you're using `PreparedStatement` objects in order to avoid falling into the traps of doing simple string concatenation and opening yourself to injection attacks (see Item 61). So you get tempted to either blow off the possibility of the injection attack—bad idea, mate—or you start cheating in other ways, like `SELECT * FROM table`, which pulls every column in the table across, rather than just the data you're interested in, which means you're wasting bandwidth. That might not be an issue on your development machine or network, but in production that could easily max out the pipe if 1,000 users execute that command simultaneously.[2] If you don't need it, you shouldn't ask for it across the wire. So what are self-respecting programmers, seeking to do the right thing while saving their poor hands from carpal tunnel syndrome, to do? (And if you're thinking this is a pretty frivolous example, take another approach to the problem: What happens if we change the table definitions, which in turn means changing the SQL statements now scattered all over the codebase?)

One approach is to keep a table-oriented view of the database but put a bit more scaffolding between you and the data access technology itself, through a Table Data Gateway [Fowler, 144]. Essentially, each table becomes a class, which in turn serves as the point of access to any rows in that table:

```
public class PersonGateway
{
  private PersonGateway() { /* Singleton—can't create */ }

  public static Person[] findAll()
```

2. OK, it would have to be an awfully wide table, but you get the idea.

```
  {
    ArrayList al = new ArrayList();

    RowSet rs = MyRowSetFactory.getRowSet();
    rs.setCommand("SELECT first_name,last_name,age "+
                  "FROM person");
    rs.execute();
    while (rs.next())
      al.add(new Person(rs.getString(1),
                        rs.getString(2),
                        rs.getInt(3));

    return (Person[])al.toArray(new Person[0]);
  }
  public static void update(Person p)
  {
    // And so on, and so on, and so on
  }
}
```

Note that when done this way, a Table Data Gateway looks a lot like a procedural-first approach to persistence (see Item 42); the key difference is that a Table Data Gateway is per-table and focuses exclusively on manipulating that table's data, whereas a procedural-first approach provides a single point of access for all persistence logic in a table-agnostic way (since the underlying data store may not even be relational tables anymore).

"This works for tables," you may be tempted to say, "but I need to do a lot of queries that aren't restricted to a single table—what then?" In fact, thanks to the closure property of relational data access mentioned earlier, the Table Data Gateway can be extended to be a new variant, say, the Query Data Gateway, where instead of wrapping around a single table, we wrap it around a query instead:

```
public class ChildrenGateway
{
  private ChildrenGateway() { }
    // Singleton, can't create
```

```
  public static Person[] findKidsForPerson(Person p)
  {
    ArrayList al = new ArrayList();

    RowSet rs = MyRowSetFactory.getRowSet();
    rs.setCommand("SELECT first_name,last_name,age "+
                  "FROM person p, parent_link pp "+
                  "WHERE p.id = pp.child_id "+
                      "AND p.last_name=?");
    rs.setInt(1, p.getPersonID());
    rs.execute();
    while (rs.next())
      al.add(new Person(rs.getString(1),
                        rs.getString(2),
                        rs.getInt(3));

    return (Person[])al.toArray(new Person[0]);
  }
}
```

Ironically, it turns out that this approach was commonly used within the relational world, long before we upstart Java programmers came along, so databases support this intrinsically: it's called a *view*. The database basically pretends that a query looks and acts just like a table:

```
CREATE VIEW children AS
  SELECT first_name, last_name, age
  FROM person p, parent_link pp
  WHERE p.id = pp.child_id
```

This creates a pseudo-table called `children` that contains all `Person` objects that have parents. We then issue queries against `children` by restricting against `last_name`.

What's the advantage here? All we've done is create something that looks like a table that isn't, and won't that create some kind of performance scariness inside the database? For some of the older database products, yes—but that problem was pretty much corrected some years ago. While there are some restrictions on views, such as on updates to a view—some database products don't allow updates at all, others allow updates only to one table if there's a multi-table join in the view, and so on—as a means

by which to make queries more readable, and in some cases more efficient, views are a powerful tool.

If none of these approaches seem particularly what you're looking for because they still seem like too much work and are too awkward to work with from within Java, another approach is to make use of the recently standardized SQL/J, or "embedded SQL for Java," Specification recently approved as part of the SQL-99 Specification. As with other embedded SQL technologies, SQL/J allows a programmer to write SQL statements directly within the Java code, which is then preprocessed by a SQL/J preprocessor, turned into regular Java code, and fed to javac as a normal compilation step. The SQL/J preprocessor takes care of any of the call-level interface logic in your Java code—you just focus on writing the SQL when you want to talk to the database and on Java when you want to do anything else.

The easiest way to use SQL/J is to write static SQL statements embedded in the code directly, known as *SQL/J clauses,* as shown here:

```
public static float averageAgeOfFamily(String lastName)
  throws Exception
{
  #sql iterator Ages (int individualAge);

  Ages rs;
  #sql rs =
    { SELECT age FROM person
      WHERE last_name = :lastName };

  int totalAge = 0;
  int numPersons = 0;
  while (rs.next())
  {
    numPersons++;
    totalAge += rs.individualAge();
  }

  return ((float)totalAge) / ((float)numPersons);
}
```

It looks and feels like a standard Java method, with the exception of the `#sql` blocks, setting off the code that will be handled by the preprocessor,

and the `Ages` type introduced by the `#sql iterator` clause early in the method body. This isn't the place to discuss the ins and outs of SQL/J syntax; download the reference implementation and documentation from http://www.sqlj.org.

The thing to recognize here is that SQL/J essentially hides almost all of the data access plumbing from sight, leaving only the barest minimum of scaffolding in place to make it easy to write the SQL itself—in many respects, it's the absolute inverse of an objects-first technology like JDO, which tries so hard to hide the relational access from the programmer's perspective. On top of this "get right to the relational data access" idea, SQL/J also offers a major improvement over any other data access technology to date: if the database schema is available at the time of compilation, a SQL/J preprocessor/translator can actually check, in compile time, the SQL in the Java code against the schema, catching typos and syntax errors without having to rely on runtime `SQLException` instances to find them. This alone is a compelling reason to take a hard look at SQL/J.

One major drawback to SQL/J, however, is its relative lack of support within the J2EE community. Although the J2EE and EJB Specifications do mention it by name as a possible data access technology layer, it receives almost no attention beyond that. Oracle is the only major database vendor to have a SQL/J implementation available as of this writing, and there seems to be little support from the community as a whole, which is a shame because in many respects this is the epitome of a relational-first persistence approach.

As with the objects-first persistence approach, the relational-first approach has its share of strengths and weaknesses. It exposes the power and grace that SQL itself encompasses, but at the expense of having to abandon (to at least a certain degree) the beauty of the object model in obtaining and updating data. It means having to write code that takes data out of the relational API, be that JDBC or SQL/J, and puts it into your object model, but the tradeoff is that you get to control the exact SQL produced, which can be a powerful optimization because not all SQL is created equal (see Item 50).

Item 42: Use procedural-first persistence to create an encapsulation layer

Prior to the object-oriented takeover of the programming language community, prior to the almost universal sweep of data storage options in the late 1970s and early 1980s that made relational databases ubiquitous, another approach quickly made its way through the ranks of IT.

Instead of directly accessing the data storage layer through SQL, middleware products (TP Monitors and the like) provided well-known entry point routines, what we would call *procedures* today. These procedures accepted a set of input (parameters) and guaranteed that they would store the data to the system, returning some kind of result indicating success or failure of the action. The details of the actual storage schema and mechanism were hidden behind these opaque walls of the procedure's entry point—so long as the middleware layer ensured that the data was stored appropriately (and could be retrieved, usually via another procedure), no further knowledge was necessary. Later, these procedures were extended to also provide certain kinds of processing, such as providing transactional semantics as a way to guarantee that the entire procedure could execute with ACID properties in place. Even later this processing was made available directly within the database, via what we now call *stored procedures*.

In the beginning, the goal was to provide an encapsulation layer protecting the raw data from programs manipulating it, which eventually set the stage for transactional processing, and later, the database management system itself. In many respects, SQL was an outgrowth of this idea, providing a declarative language that allowed operators to simply say what data they were interested in, based on a set of constraints expressed as part of the SQL statement. This approach, aside from being extremely powerful in the right hands, also proved difficult for programmers familiar with procedural (and later, object-oriented) languages to understand.

Like other declarative languages (Prolog and the more modern XSLT), SQL requires that users state only what data they're interested in, not how to approach obtaining that data. This fits well with the overall "flow" of the relational model but feels awkward to procedural programmers first learning SQL—they keep wanting to "start with this table over here, then go get this data element over here based on . . . ," and so on. Declarative and procedural languages are two very different beasts (just as objects and

relations are). This has prompted relational experts like Joe Celko to state, "The single biggest challenge to learning SQL programming is unlearning procedural programming" [Henderson03, 1].

Rather than adopt one of the other models, then, what works best in some cases is to simply bury the details of persistence behind another layer of software, such that the messy details of doing the persistence can be hidden away from the programmer's view—true encapsulation. Doing this can be as simple as the following code:

```
public class DataManager
{
  private DataManager()
  { /* Prevent accidental creation */ }

  public Person findPersonByLastName(String lastName)
  {
    // Open a Connection, create a Statement, execute
    // SELECT * FROM person p WHERE
    // p.last_name = (lastName)
    // Extract data, put into new Person instance,
    // return that
  }
  public void updatePerson(Person p)
  {
    // Open a Connection, create a Statement, execute
    // UPDATE person p SET . . .
  }
}
```

Note that the `DataManager` is a Singleton but runs locally in the client's virtual machine; it relies on the database access layer (in this case, JDBC) to protect it from any concurrency concerns in order to avoid being a Remote Singleton (see Item 5).

While this makes the `DataManager` a relatively large class to work with, it does have a number of advantages that make it an attractive option.

- Data access experts can tune the data access logic (which we'll assume is SQL) as necessary to achieve optimal results.
- If necessary, changing data storage systems means changing this one class. If we want to change from a relational back end to an

OODBMS, client code doesn't have to change to do the persistence work (even if it is potentially unnecessary, depending on the OODBMS in use and its underlying persistence semantics).

- Since we know that all the persistence logic for this request is bracketed within this function, clients don't have to worry about transactional semantics or transactions being held open any longer than necessary; this is important to minimize lock windows (see Item 29).
- We can use whatever data access APIs make the most sense for implementing the methods in this one class without changing any client code. If we start by using JDBC and find that JDBC is too low-level to use effectively (not an uncommon opinion), we can switch to SQL/J for this one class only, and clients are entirely ignorant of the change. If we find that SQL/J is easier to work with but doesn't generate optimal JDBC-based access, we can switch back with impunity.
- We could keep back-end portability by making `DataManager` an interface and creating `DataManager` derivatives that specialize in doing storage to one or more different back-end storage layers—an `OracleDataManager`, an `HSQLDataManager` (for in-process relational data storage), a generic `JDBCDataManager` (for generic nonoptimized JDBC-SQL access), and so on. This allows us to tune for a particular back end without sacrificing overall system portability, but at the expense of a lot of work to build each one, including maintenance when the `DataManager` interface needs to change.
- New access requires modifying the centralized class. This may seem like a disadvantage at first—after all, who wants to go back and modify code once it's "done"?—but the fact that somebody wants a new query or update gives the `DataManager` developer an opportunity to revisit the data access code and/or the actual storage of the data. We can use this to add a few optimizations based on the new usage, such as creating new indexes in the relational model. It also serves as a sanity check to prevent looking up data by invalid criteria ("Well, since everybody knows that only men are programmers, I was going to do the programmer lookup by adding criteria to restrict by `gender=male`; that should speed up the query, right?").

In essence, the `DataManager` class serves as our sole point of coupling between the data storage tier and the Java code that wants to access it.

Procedural access isn't always done at the Java language level, however. Most commercial databases offer the ability to execute procedural code directly within the database itself, a stored procedure in DBMS

nomenclature. While all the major database vendors offer vendor-specific languages for writing stored procedures, many are starting to support the idea of writing Java stored procedures (an unfortunate overload of the acronym JSP), meaning Java programmers don't have to learn another language. However, one of the brightest advantages of using the stored-procedure-as-persistence model is that we can pass the implementation of the stored procedure to the database staff, leaving the entire data model in their hands to be designed, implemented, and tuned as necessary. As long as the definitions of the stored procedures don't change (we'll get runtime `SQLException` instances if they do) and the stored procedure implementations do what we expect them to (like store or retrieve the data), we can remain entirely ignorant of the underlying data model—which, if you're as good at relational design and execution as I am (that is to say, not at all), is a good thing.

By the way, if you're thinking that the procedural model is a horrible idea, and you would never consider using anything like it, stop and have another look at the Session Façade [Alur/Crupi/Malks, 341], Domain Store [Alur/Crupi/Malks, 516], and other data access patterns. In order to avoid round-trips to the database, recall that current EJB "best practice" thinking has clients calling session beans, passing either Data Transfer Objects [Fowler, 401] or other data in bulk (see Item 23) as parameters to the session bean method, to be extracted and inserted into the local entity bean for storage in the database. This is no different than the `DataManager` presented earlier, except that now `DataManager` is a session bean. The same will be true of just about any Web service–based data access model.

Item 43: Recognize the object-hierarchical impedance mismatch

XML is everywhere, including in your persistence plans.

Once we'd finally gotten around to realizing that XML was all about data and not a language for doing markup itself as HTML was, industry pundits and writers started talking about XML as the logical way to represent objects in data form. Shortly thereafter, the thought of using XML to marshal data across the network was introduced, and SOAP and its accompanying follow-up Web Service specifications were born.

The problem is that XML is intrinsically a hierarchical way to represent data—look at the XML Infoset Specification, which requires that data be

well formed, meaning the elements in an XML document must form a nice tree of elements (each element can have child elements nested within it, each element has a single parent in which it's nested, with the sole exception of the single "root" node that brackets the entire document, and so on). This means that XML is great for representing hierarchical data (hence the title of this item), and assuming your objects form a neat hierarchy, XML is a natural way to represent that data (hence the natural assumption that XML and objects go hand in hand).

But what happens when objects don't form nice, natural trees?

Hierarchical data models are not new; in fact, they're quite old. The relational data model was an attempt to find something easier to work with than the database systems of the day, which were similar in concept, if not form, to the hierarchical model we see in XML today. The problem with the hierarchical model at the time was that attempting to find data within it was difficult. Users had to navigate the elements of the tree manually, leaving users to figure out "how" instead of focusing on "what"—that is, how to get to the data, rather than what data they were interested in.

With the emergence of XML (and the growing interest in "XML databases," despite the inherent ambiguity in that term), it would seem that hierarchical data models are becoming popular once again. While a full discussion of the implications of a hierarchical data model are beyond the scope of this book, it's important to discuss two things here: when we're likely to use a hierarchical data model in J2EE, and what implications that will have for Java programmers.

While the industry currently doesn't recognize it, mapping objects to XML (the most common hierarchical storage model today) is not a simple thing, leading us to wonder whether an *object-hierarchical impedance mismatch*—in other words, a mismatch between the free-form object model we're all used to and the strictly hierarchical model the XML Infoset imposes—is just around the corner.[3] In fact, given that we now have vendors offering libraries to map objects to XML for us, as well as the more recent Java API for XML Binding (JAXB) standard to help

3. Let's not even consider the implications of objects stored in relational databases being transformed into XML: the idea of an object-relational-hierarchical impedance mismatch is enough to move the strongest programmer to tears.

unify the various implementations that do so, it may be fair to infer that mapping objects to XML and back again isn't as simple as it seems—granted, simple object models map to XML pretty easily, but then again, simple object models map pretty easily to relational tables, too, and we all know how "easy" it is to do object-relational mapping.

Much of the problem with mapping objects to a hierarchical model is the same problem that occurs when mapping objects to a relational model: preserving object identity. To understand what I mean, let's go back for a moment to the same `Person` object we've used in previous items:

```
public class Person
{
  // Fields public just for simplicity
  //
  public String firstName;
  public String lastName;
  public int age;

  public Person(String fn, String ln, int a)
  { firstName = fn; lastName = ln; age = a; }
}
```

Again, simple and straightforward, and it's not overly difficult to imagine what an XML representation of this object would look like:

```
<person>
  <firstName>Ron</firstName>
  <lastName>Reynolds</lastName>
  <age>30</age>
</person>
```

So far, so good. But now, let's add something that's completely reasonable to expect within an object-oriented model but completely shatters a hierarchical one—cyclic references:

```
public class Person
{
  public String firstName;
  public String lastName;
  public int age;
  public Person spouse;
```

```
  public Person(String fn, String ln, int a)
  { firstName = fn; lastName = ln; age = a; }
}
```

How do you represent the following set of objects?

```
Person ron = new Person("Ron", "Reynolds", 31);
Person lisa = new Person("Lisa", "Reynolds", 25);
ron.spouse = lisa;
lisa.spouse = ron;
```

A not-unreasonable approach to serializing `ron` out to XML could be done by simply traversing the fields, recursively following each object as necessary and traversing its fields in turn, and so on; this is quickly going to run into problems, however, as shown here:

```
<person>
  <firstName>Ron</firstName>
  <lastName>Reynolds</lastName>
  <age>31</age>
  <spouse>
    <person>
      <firstName>Lisa</firstName>
      <lastName>Reynolds</lastName>
      <age>25</age>
      <spouse>
        <person>
          <firstName>Ron</firstName>
          <lastName>Reynolds</lastName>
          <age>31</age>
          <spouse>
            <!-- Uh, oh . . . -->
```

As you can see, an infinite recursion develops here because the two objects are circularly referencing one another. We could fix this problem the same way that Java Object Serialization does (see Item 71), by keeping track of which items have been serialized and which haven't, but then we're into a bigger problem: Even if we keep track of identity within a given XML hierarchy, how do we do so across hierarchies? That is, if we serialize both the `ron` and `lisa` objects into two separate streams (perhaps as part of a JAX-RPC method call), how do we make the deserialization logic aware of the fact that the data referred to in

the `spouse` field of `ron` is the same data referred to in the `spouse` field of `lisa`?

```
String param1 = ron.toXML(); // Serialize to XML
String param2 = lisa.toXML(); // Serialize to XML
sendXMLMessage("<parameters>" + param1 + param2 +
               "</parameters>");

/* Produces:
param1 =
<person id="id1">
  <firstName>Ron</firstName>
  <lastName>Reynolds</lastName>
  <age>31</age>
  <spouse>
    <person id="id2">
      <firstName>Lisa</firstName>
      <lastName>Reynolds</lastName>
      <age>25</age>
      <spouse><person href="id1" /></spouse>
    </person>
  </spouse>
</person>
param2 =
<person id="id1">
  <firstName>Lisa</firstName>
  <lastName>Reynolds</lastName>
  <age>25</age>
  <spouse>
    <person id="id2">
      <firstName>Ron</firstName>
      <lastName>Reynolds</lastName>
      <age>25</age>
      <spouse><person href="id1" /></spouse>
    </person>
  </spouse>
</person>
 */

// . . . On recipient's side, how will we get
// the spouses correct again?
```

(By the way, this trick of using `id` and `href` to track object identity is not new. It's formally described in Section 5 of the SOAP 1.1 Specification, and as a result, it's commonly called *SOAP Section 5 encoding* or, more simply, *SOAP encoding*.) We're managing to keep the object references straight within each individual stream, but when we collapse the streams into a larger document, the two streams have no awareness of one another, and the whole object-identity scheme fails. So how do we fix this?

The short but brutal answer is, we can't—not without relying on mechanisms outside of the XML Infoset Specification, which means that schema and DTD validation won't pick up any malformed data. In fact, the whole idea of object identity preserved by SOAP Section 5 encoding is entirely outside the Schema and/or DTD validator's capabilities and has been removed in the latest SOAP Specification (1.2). Cyclic references, which are actually much more common in object systems than you might think, will break a hierarchical data format every time.

Some will point out that we can solve the problem by introducing a new construct into the stream that "captures" the two independent objects, as in the following code:

```
<marriage>
  <person>
    <!-- Ron goes here -->
  </person>
  <person>
    <!-- Lisa goes here -->
  </person>
</marriage>
```

But that's missing the point—in doing this, you've essentially introduced a new data element into the mix that doesn't appear anywhere in the object model it was produced from. An automatic object-to-XML serialization tool isn't going to be able to make this kind of decision, and certainly not without some kind of developer assistance.

So what? It's not like we're relying on XML for data storage, for the most part—that's what we have the relational database for, and object-relational mapping layers will take care of all those details for us. Why bother going down this path of object-hierarchical mapping?

If you're going to do Web Services, you're going to be doing object-hierarchical mapping: remember, SOAP Section 5 encoding was created

to solve this problem because we want to silently and opaquely transform objects into XML and back again without any work on our part. And the sad truth is, just as object-relational layers will never be able to silently and completely take care of mapping objects to relations, object-hierarchical layers like JAXB or Exolab's Castor will never be able to completely take care of mapping objects to hierarchies.

Don't think that the limitations all go just one way, either. Objects have just as hard a time with XML documents, even schema-valid ones, as XML has with object graphs. Consider the following schema:

```
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'
    xmlns:tns='http://example.org/product'
    targetNamespace='http://example.org/product' >
  <xsd:complexType name='Product' >
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name='produce'
                     type='xsd:string'/>
        <xsd:element name='meat' type='xsd:string' />
      </xsd:choice>
      <xsd:sequence minOccurs='1'
                    maxOccurs='unbounded'>
        <xsd:element name='state'
                     type='xsd:string' />
        <xsd:element name='taxable'
                     type='xsd:boolean'/>
      </xsd:sequence>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name='Product' type='tns:Product' />
</xsd:schema>
```

Here is the schema-valid corresponding document:

```
<groceryStore xmlns:p='http://example.org/product'>
  <p:Product>
    <produce>Lettuce</produce>
    <state>CA</state>
    <taxable>true</taxable>
    <state>MA</state>
    <taxable>true</taxable>
```

```
    <state>CO</state>
    <taxable>false</taxable>
  </p:Product>
  <p:Product>
    <meat>Prime rib</meat>
    <state>CA</state>
    <taxable>false</taxable>
    <state>MA</state>
    <taxable>true</taxable>
    <state>CO</state>
    <taxable>false</taxable>
  </p:Product>
</groceryStore>
```

Ask yourself this question: How on earth can Java (or, for that matter, any other traditional object-oriented language, like C++ or C#) represent this repeating sequence of element `state`/`taxable` pairs, or the discriminated union of two different element types, `produce` or `meat`? The closest approximation would be to create two subtypes, one each for the `produce` and `meat` element particles, then create another new type, this time for the `state`/`taxable` pairs, and store an array of those in the `Product` type itself. The schema defined just one type, and we have to define at least four in Java to compensate.

Needless to say, working with this schema-turned-Java type system is going to be difficult at best. And things get even more interesting if we start talking about doing derivation by restriction, occurrence constraints (`minOccurs` and `maxOccurs` facets on schema compositors), and so on. JAXB and other Java-to-XML tools can take their best shot, but they're never going to match schema declarations one-for-one, just as schema and XML can't match objects one-for-one. In short, we have an impedance mismatch.

Where does this leave us?

For starters, recognize that XML models hierarchical data well but can't effectively handle arbitrary object graphs. In certain situations, where objects model into a neat hierarchy, the transition will be smooth and seamless, but it takes just one reference to something other than an immediate child object to seriously throw off object-to-XML serializers. Fortunately, strings, dates, and the wrapper classes are usually handled in a pretty transparent manner, despite their formal object status, so that's

not an issue, but for anything else, be prepared for some weird and semi-obfuscated results from the schema-to-Java code generator.

Second, take a more realistic view of what XML can do for you. Its ubiquity makes it a tempting format in which to store all your data, but the fact is that relational databases still rule the roost, and we're mostly going to use XML as an interoperability technology for the foreseeable future. Particularly with more and more RDBMS vendors coming to XML as a format with which to describe data, the chances of storing data as XML in an "XML database" are slight. Instead, see XML as a form of "data glue" between Java and other type systems, such as .NET and C++.

A few basic principles come to mind, which I offer here with the huge caveat that some of these, like any good principles, may be sacrificed if the situation calls for it.

- *Use XML Schema to define your data types*. Just as you wouldn't realistically consider doing an enterprise project storing data in a relational database without defining relational constraints on your data model, don't realistically consider doing enterprise projects storing data in XML without defining XML data types. Having said that, however, at times, a more flexible model of XML will be useful, such as when allowing for user extensions to the XML data instances. Be prepared for this by writing your types to have extension points within the type declarations, via judicious use of `any`-type elements. And, although this should be rare within the enterprise space, in some cases some XML data needs to be entirely free-form in order to be useful, such as Ant scripts. In those situations, be strong enough to recognize that you won't be able to (or want to) have schema types defined and that they will require hand-verification and/or parsing.
- *Prefer a subset of the schema simple types*. XML Schema provides a rich set of simple types (those that most closely model primitive types in Java), such as the `yearMonth` and `MonthDay` types for dates, but Java has no corresponding equivalent—a schema-to-Java toll will most likely model both of those, as well as many others, as a simple integer field. Unfortunately, that means you can store anything you want into that field, thus losing the type definition intended by the schema document in the first place. To avoid this situation, prefer to stick to the types in XSD schema that most closely model what Java (and .NET and any other language you may end up talking to) can handle easily.

- *Use XML Schema validating parsers to verify instances of your schema types when parsing those instances*. The parser will flag schema-invalid objects, essentially acting as a kind of data-scrubbing and input-validating layer for you, without any work on your part. This will in turn help enforce that you're using document-oriented approaches to your XML types, since straying from that model will flag the validator. Be aware, though, that schema-validating parsers are woefully slow compared to their non-schema-validating counterparts. More importantly, schema-validating parsers will only be able to flag schema-invalid objects, and if you're taking an object-based approach to your XML types that uses out-of-band techniques (like SOAP encoding does), schema validators won't pick it up, so you'll know there's a problem only when the parser buys off on the object but your code doesn't. That's a hard scenario to test for.
- *Understand that type definitions are relative.* Your notion of what makes a `Person` is different from my notion of what makes a `Person`, and our corresponding definitions of `Person` (whether in object type definitions, XML Schema, or relational schema) differ accordingly. While some may try to search for the Universal Truth regarding the definition of `Person`, the fact is that there really isn't one—what's important to your organization about `Person` is different from what's important to my organization, and no amount of persuasion on your part is going to change that for me, and vice versa. Instead of going down this dead-end road, simply agree to disagree on this, and model schema accordingly if the documents described by this schema are to be used by both of us. In other words, use schema to verify data being sent from one system to another, rather than trying to use it to define types that everybody agrees on.
- *Avoid HOLDS-A relationships*. As with the `Person` example, instances that hold "references" to other instances create problems. Avoid them when you can. Unfortunately, that's a lot easier said than done. There is no real way to model `Person` in a document-oriented fashion if `Person` needs to refer to the spouse and still preserve object identity. Instead, you're going to have to recognize that the `Person`'s spouse is an "owned" data item—so instead of trying to model it as a standalone `Person`, just capture enough data to uniquely identify that `Person` from some other document (much as a relational table uses a foreign key to refer to a primary key in another

table). Unfortunately, again, XML Schema can't reflect this[4] and it will have to be captured in some kind of out-of-band mechanism; no schema constraints can cross documents, at least not as of this writing.

Most importantly, make sure that you understand the hierarchical data model and how it differs from relational and object models. Trying to use XML as an objects-first data repository is simply a recipe for disaster—don't go down that road.

Item 44: Use in-process or local storage to avoid the network

Most often, when J2EE developers begin the design and architectural layout of the project, it's a foregone conclusion that data storage will be done in a relational database running on a machine somewhere in the data center or operations center.

Why?

There's no mystical, magical reason, just simple accessibility. We want the data to be accessible by any of the potential clients that use the system.

In the old days, before *n*-tier architectures became the norm, clients connected directly to servers, so the data needed to be held directly on the server, in order to allow all the clients access to all the data; networks hadn't yet achieved widespread ubiquity, and wireless access technology, which makes networking even simpler, hadn't arrived. Connecting machines to a network was a major chore, and as a result, the basic concepts of peer-to-peer communication were reserved for discussions at the lowest levels of the networking stack (the IP protocol, for example).

In time, we realized that putting all of the data on a central server had the additional advantage of putting major load on the server, thereby taking it off the clients. Since the server was a single machine, it was much more cost-effective (so we believed) to upgrade or replace that single machine, rather than all of the clients that connected to it. So not until well after we

4. If this XML document is a collection of `Person` instances, and the spouse is guaranteed to be within this collection someplace, that's a different story, but that also changes what we're modeling here and becomes a different problem entirely.

had established the idea of the centralized database did we started putting our databases on the heaviest iron we could find, loading them up with maximal RAM footprints and huge gigabyte (and later, terabyte) drive arrays.

The point of this little digression into history is that the centralized, remote database server exists simply to provide a single gathering point for data, not because databases *need* to run on servers that cost well into five, six, or seven digits. We put the data on the server because it was (a) a convenient place to put it, and (b) an easy way to put processing in one central place for all clients to use without having to push updates out (zero deployment), and (c) a way to put the data close to the processing (see Item 4).

Transmitting all this data across the wire isn't cheap, however, nor is it without its own inherent share of problems. It's costly both in terms of scalability and performance (since each byte of bandwidth consumed to transfer data around the network is a byte of bandwidth that can't be used for other purposes), and the time it takes to shuffle that data back and forth is not trivial, as Item 17 describes. So, given that we put data on the centralized database in order to make it available to other clients and that it's not cheap to do the transfer, don't put data on the remote database unless you have to; that is, don't put any data on the remote database unless it's actually going to be shared across clients.

In such cases, running a relational database (or another data storage technology—here we can think about using an object database or even an XML database if we choose to) inside the same process as the client application can not only keep network round-trips to a minimum but also keep the data entirely on the local machine. While running Oracle inside of our servlet container is probably not going to happen any time soon, running an all-Java RDBMS implementation is not so far-fetched; for example, Cloudscape offers this functionality, as do PointBase and HSQLDB (an open-source implementation on Sourceforge), essentially becoming a database masquerading as a JDBC driver. Or, if you prefer an object-based approach, Prevayler, another open-source project, stores any Java object in traditional objects-first persistence fashion. If you'd rather see the data in a hierarchical fashion, Xindice is an open-source XML database from the Apache Group.

One simple in-process data storage technique is the `RowSet` itself. Because the `RowSet` is entirely disconnected from the database, we can

create one around the results of a query and keep that `RowSet` for the lifetime of the client process without worrying about the scalability effects on the database. Because the `RowSet` is Serializable, we can store it as is without modification into any `OutputStream`, such as a file or a `Preferences` node; in fact, if the `RowSet` is wrapped around configuration data, it makes more sense to store it in a `Preferences` node than in a local file in some ways (see Item 13). It won't scale to thousands of rows, it won't enforce relational integrity, but if you need to store that much data or to put relational constraints around the local data, you probably want a "real" database, like HSQLDB or a commercial product that supports "embedding" in a Java process.

There's another side to this story, however—the blindingly obvious realization that a remote database requires a network to reach it. While this may seem like an unnecessary statement, think about that for a moment, and then consider the ubiquitous sales-order application tied into the inventory database for a large manufacturing company. We release the application onto the salesperson's laptop, and off the salesperson goes to pay a visit to the VP of a client company. After a little wining, dining, and 18 holes on a pretty exclusive golf course, the VP's ready to place that million-dollar order. The salesperson fires up the laptop, goes to place the order, and to his horror, gets an error: "database not found." Sitting in the golf club's posh restaurant, our plucky salesperson suddenly pales, turns to the VP, and says, "Hey, um, can we go back to your office so I can borrow your network?"

Assuming the VP actually lets our intrepid salesperson make use of her network to connect back to the salesperson's home network via VPN, assuming the salesperson knows how to set that up on his laptop, assuming the IT staff at home has opened a VPN in the corporate network, and assuming the application actually works with any speed over the VPN connection, the salesperson's credibility is taking a pretty serious beating here. On top of this, the VP has every reason to refuse the salesperson the network connection entirely—it's something of a security risk, after all, to let foreign machines onto the network behind the firewall. And the IT staff at home has every reason to keep the database as far away from the "outside world" as possible, VPN or no VPN. By the time the salesperson returns to the home office to place the order (scribbled on the napkin from the posh restaurant), the VP may have changed her mind, or the salesperson may have forgotten to get some important detail onto the napkin, and so on. (It's for reasons like these that salespeople are

trained to place the order as quickly as possible as soon as the customer approves it.)

In some circles, this is called the *traveling salesman problem* (not to be confused with the problem-solving cheapest-way-to-travel version of the problem commonly discussed in artificial intelligence textbooks). The core of the problem is simple: you can't always assume the network will be available. In many cases, you want to design the application to make sure it can run without access to the network, in a disconnected mode. This isn't the same as offering up a well-formatted, nicely handled `SQLException` when the router or hub hiccups; this is designing the application to take the salesperson's order on the standalone laptop with no network anywhere in sight anytime soon. When designing for this situation, ask yourself how the application will behave when users are 37,000 feet in the air, trying to sell widgets to somebody they just met on the airplane.

One of the easiest ways to accommodate this scenario is to keep a localized database running, with a complete data dump of the centralized database on the same machine as the client (which, by the way, should probably be a rich-client application since the network won't be there to connect to the HTTP server, either—see Item 51). When the machine is connected via the network to the remote database, it updates itself with the latest-and-greatest data from the centralized database. Not necessarily the complete schema, mind you, but enough to be able to operate without requiring the remote database. For example, in the hypothetical sales application, just the order inventory, detail information, and prices would probably be enough—a complete dump of open sales, their history, and shipping information probably doesn't need to be captured locally. Then, when new orders are placed, the application can update the local tables running on the same machine.

Some of you are undoubtedly cringing at the entire suggestion. "Not connected to the remote database? Inconceivable! How do we avoid data integrity errors? That's why we centralized the database in the first place! After all, if there are only 100 widgets left, and both Susan and Steve sell those last 100 widgets to different clients via their laptop systems, we have an obvious concurrency issue when we sync their data up against the centralized system. Any system architect knows that!"

Take a deep breath. Ask yourself how you're going to handle this scenario anyway, because whether it happens at the time the salesperson places the

order or when the order goes into the database, the same problem is still happening. Then go take a look at Item 33 as one way to solve it.

Normally, in a connected system, inventory checking occurs when the salesperson places the order—if the requested number of widgets isn't available on the shelves, we want to key the order with some kind of red flag and either not let the order be placed or somehow force the salesperson to acknowledge that the inventory isn't available at the moment. In the connected version of the application, this red flag often comes via a message box or alert window—how would this be any different from doing it later, when the salesperson is synchronizing the data against the centralized system? In fact, this may be a preferable state of affairs, because that alert window could potentially ruin the sale. If the VP sees the message she may rethink placing the order: "Well, I'll bet your competitor has them in stock, and we need these widgets right now," even if "right now" means "we can use only 50 a week." Instead, when the red flag goes off back at the office, the salesperson can do a little research (difficult, if not impossible, to do sitting in the client VP's office) before calling the customer to tell them the news. ("Well, we had only 50 widgets here in the warehouse, but Dayton has another 50, and I'm having them express-mailed to you, no charge.")

While the remote database has obvious advantages for enterprise systems—after all, part of our working definition of an enterprise system is that it manages access to resources that must be shared, which implies centralization—the remote database doesn't necessarily serve as the best repository for certain kinds of data (like per-user settings, configuration options, or other nonshared data) or for certain kinds of applications, particularly those that desire or need to run in a disconnected fashion.

Item 45: Never assume you own the data or the database

Remember that one of our fundamental assumptions about enterprise software (from Chapter 1) is that an enterprise software system is one in which some or all of its resources are shared. In fact, the core of the enterprise system—the data—is most frequently the shared resource in question. Most of the time, we assume that the extent of the sharing occurs among the various users of the enterprise system. However, it doesn't stop there—many enterprise databases count other enterprise systems as part

of their userbase, and unfortunately those "users" have much more stringent requirements than usual.

Some of these other systems making use of your database aren't strangers; it's not uncommon, for example, to design or purchase a reporting engine to allow end users to create their own reports from the database. This is a good thing; without that, the need to create the logic and nicely formatted output will fall into developers' laps. While constantly changing reports to keep up with user whims is one way to stay employed for the rest of your life, it also means you never get to do anything else on the project, and that's not my idea of an interesting job. Those reporting engines typically work directly against the database schema, and here's where we need to be careful.

If multiple systems or agents share a resource, the format or schema of that resource cannot change at the whim of one of those systems or agents without causing a ripple effect through the rest. Alterations to the schema, excessive database locks, and relocation of the database instance are changes that may seem localized to just your codebase but will in turn create problems and issues with these other systems. In fact, this is what keeps legacy systems alive for so long—it's extremely difficult to track down every client for a given resource or repository within the company (particularly large companies), much less figure out all the data dependencies of a given client against a single system. As a result, it's far easier to just leave the legacy system in place and create adapters, translators, and proxies than to replace the legacy system with something new.

To the J2EE developer, this has some sobering, if not chilling, implications: you do not own your database, nor do you own the data within it. Even if you are building the database entirely from scratch for this project, even if the project seems to be tucked away in a little corner of the business where nobody else will have any interest in it, you don't own that database. It's only a matter of time before somebody else within the company hears about your little system and wants to gain access to that data, so you start setting up back-end data feeds, direct connections, and so on. Or worse, somebody else on your team sets them up without telling you. Before long, you make a schema change, and people you've never even heard of, much less met, are howling into your phone demanding to know why you broke their application.

The conclusion here is clear: your schema, once designed, is in far greater danger of being locked in than your code. This is where having an

encapsulation layer between the users and the database (as discussed in Item 42) becomes so critical—by forcing clients (any clients, including your own code) to go through some kind of encapsulatory barrier, such as a stored procedure layer or Web Service endpoint layer, to gain access to the data, you essentially plan for the future and the fact that others will need access to the data even as you modify the underlying schema and data description definitions.

This also means that the database schema shouldn't be optimized for the J2EE case: be careful when building your schema not to tie it too closely to your object model, be extremely careful about storing serialized Java objects in Binary Large Object (BLOB) fields since doing so means that data cannot be used as part of a SQL query (unless your database supports it, in which case the query will perform extremely slowly), and don't store "magic values" in the database. For example, although it might be convenient to store Java `Date` objects as long integers (the underlying value held in a `java.util.Date`, accessible via `getTime`, and the only nondeprecated way to construct a `java.util.Date`), other languages will have little to no idea what to do with a value like that, and converting it into a date value acceptable to them is difficult. Some databases may try to implicitly convert an integer value into a date, for example, but how they convert the value will be entirely different for each database. Despite the greater work in Java, it's far better to store the date in the database in either a string-based format or as an ANSI-standard `DATETIME`.

In fact, because your database is likely to become a shared database fairly quickly, and because J2EE isn't slated to take over the entire world any time soon (the .NET and PHP programmers may have a few words for those who think it will), you'll generally prefer to put whatever constraints you can into the database, rather than relying on J2EE code to enforce restrictions. For example, given the choice between checking to see whether an incoming string is fewer than 40 characters long before storing it to the database, go ahead and simply rely on database integrity constraints on the table to check the length of the string, rather than doing it in J2EE and setting the column's size to be something larger than that. Since the database is going to go through the size-constraint check anyway, why do it twice? Similarly, model any sort of relationship between tables/entities in the database directly within the database, using foreign-key constraints to ensure that both sides of the constraint are in fact present within the database.

It may seem easier, at first blush, to go ahead and encode this sort of logic directly within your object model and/or domain logic—after all, Java is probably your first language, and it's a natural human tendency to want to solve problems using the tools we're most comfortable with. Resist. The database provides a lot of functionality that can't be easily—or as universally—applied in your J2EE code. For example, most database products support the concept of triggers, blocks of database code that can be executed on any sort of row- or table-based access, giving you the ability to apply domain logic after the row has been inserted, updated, deleted, or whatever. Trying to build this into a J2EE application, which would have to be universally applied across all parts of the system, requires either building a complex notification mechanism into your code (keeping in mind the drawbacks of an RPC-based callback mechanism, as described in Item 20) or hand-coding directly into your domain object representations of the database entities themselves, leading you down a scary road of maintenance nightmares over time.

While it may seem convenient and entirely plausible to take shortcuts in the database layer to make it easier for you as a J2EE developer, resist the urge. Ultimately, it's just going to create more problems if you don't—when the inevitable question, "Hey, we noticed you were capturing some data we were interested in, and we want to get at it, so how can we?" comes up, it's going to be up to you to make that happen, and it'll be a lot easier to take care of at the front of the project than at the back.

Item 46: Lazy-load infrequently used data

Given the cost of a trip across the network to the remote database, it follows that we don't want to actually pull any data across the network unless we need it.

This seems like a simple idea, but it's remarkable how often it gets left behind in systems that try to hide the database from the programmer—for example, naïve object-relational mapping mechanisms often map objects to tables in a one-to-one fashion, so that any request to retrieve a particular object from the database means every column for a particular row gets pulled across to completely fill out the object in question.

In many cases, this is obviously a bad situation: consider the canonical drill-down scenario, where I want to present a list of items to the user

from which he or she selects one (or more) for a more complete examination. If a large number of items must be presented to the user in a summary view for selection, say, 10,000 persons that meet the initial search criteria, pulling each one back in its entirety means 10,000 × *N* bytes of data must be sent across the network, where *N* is the complete size of a single row from the table. If the row has foreign-key relations that must in turn be retrieved as part of this object retrieval (the `Person` has `0..*` `Address` instances associated with it, perhaps), that number easily bloats to unmanageable proportions.

For these reasons, many object-relational mechanisms choose not to retrieve all of the data for a particular row as part of a standard fetch operation but instead choose to lazy-load the data, preferring to take the additional round-trip to the database in exchange for keeping this summary request down to manageable levels. In fact, this is such a common scenario that it has been documented in several places already, most notably as the Lazy Load pattern [Fowler, 200].

Take careful note: the key is to lazy-load infrequently used data, not just data that hasn't been used yet.

Here the entity beans portion of EJB tends to fall down in catastrophic fashion, similar to the classic *N*+1 query problem. Many EJB entity bean implementers decided that since we wanted to avoid pulling back any data that wasn't going to be used immediately, when retrieving an entity bean from a `Home`-based finder method, *nothing* was going to be retrieved from the actual data store (relational database), and instead only the primary key for the row would be stored in the entity bean inside the EJB container. Then, when a client accessed a property method (`get` or `set`) for that entity bean, a separate query would be issued to retrieve or update that particular column value for that particular bean.

The dangerous part, of course, occurred if you wrote client-side code that did something like this under the hood of the entity bean:

```
PersonHome personHome = getHomeFromSomewhere();
Person person = personHome.findByPrimaryKey(1234);

String firstName = person.getFirstName();
String lastName = person.getLastName();
int age = person.getAge();
```

With such code, you were making requests like this:

```
SELECT primary_key FROM Person
    WHERE primary_key = 1234;

SELECT first_name FROM Person
    WHERE primary_key = 1234;

SELECT last_name FROM Person
    WHERE primary_key = 1234;

SELECT age FROM Person WHERE primary_key = 1234;
```

In addition to this being a phenomenal waste of CPU cycles (parsing each query, projecting a query plan for each, marshaling the returned column in a relational tuple, and so on), as well as flooding the cache to the point of irrelevancy, you're also taking a huge risk that the data doesn't change between bean method calls—remember, each of those calls represents a separate EJB-driven transaction, thus leaving open the possibility that a different client could change the same row between calls (see Item 31). For these reasons and more, two patterns were born: the Session Façade [Alur/Crupi/Malks, 341] and the Data Transfer Object [Fowler, 401].

The problem, however, isn't necessarily with the idea of lazy-loading; the problem is knowing *what* data to lazy-load. In the case of a container-managed entity bean implementation, where all details of SQL are hidden from the EJB developer, the EJB container has no way to know which data elements are more likely to be used than others, so it makes the more drastic assumption that all of them won't necessarily be used. In retrospect, this is probably a bad decision, one that's only slightly better than the other alternative left to the container implementor, that of assuming that *all* of the columns will be used. The ideal situation would be for the entity bean implementor to allow you to provide some kind of usage hints about which fields should be pulled back as part of pulling the initial bean state across and which ones would be best retrieved lazily, but such things are vendor-dependent and reduce your portability significantly (if you care—see Item 11).

If you're writing your own relational access code, such as writing a BMP entity bean or just writing plain-vanilla JDBC or SQL/J code, the decision of what to pull back falls into your lap once again. Here, as with any SQL access, you should be explicit about precisely what data you want:

nothing more, nothing less. This means avoiding SQL queries that look like this one:

```
SELECT * FROM Person
  WHERE ...
```

Although it may not seem all that important at the time you write this query, you're pulling back every column in the `Person` table when this code gets executed. If, at first, the definition of the `Person` table happens to match the exact list of data you want, it seems so much easier and simpler to use the wildcard `*` to retrieve all columns, but remember—you don't own the database (see Item 45). Several things can happen to the database definition that will create problems in your code if you use the wildcard.

- *Additional columns may get added.* Hey, updates happen, and unless you want the thankless job of going back through every line of SQL code you've ever written to make sure you're dealing with the additional columns, you're going to be pulling back data you didn't want.
- *Wildcard* `SELECT` *statements don't specify a column order.* Unfortunately, the order of columns returned in a `SELECT *` query isn't guaranteed by the SQL standard, and in some cases not even by the database product you're using—this means that your underlying JDBC code, which relies on an implicit order of columns returned, will be treated to a whole host of `SQLException` instances when trying to retrieve the `first_name` column as an `int`. Worse yet, this kind of problem won't show up in your development database because the database administrator isn't tweaking and toying with the database definitions to try to improve performance—it will only show up in production systems, leaving you scratching your head in confusion with little to go on when trying to ascertain precisely what the problem is.
- *The implicit documentation hint is lost.* Let's be honest, it's just clearer to be explicit about what you're retrieving when you list the columns. It also means one less comment in your Java code to maintain; whenever possible, it's better to create situations where code speaks for itself.

Despite the additional typing required, it's almost always preferable to list the columns explicitly. Fortunately, it's also something that's fairly easy to generate via a code-generation tool, should the opportunity arise.

Lazy-loading isn't just restricted to loading columns from the query, however. There are other opportunities to apply the same principle to larger

scales. For example, go back to the canonical drill-down scenario. In most thin-client applications, when retrieving search results, we often display only the first 10, 20, maybe 50 results to the user, depending on the size of the results window. We don't want to display to the user the complete set of results—which sometimes stretches into the thousands of items when the user provides particularly vague criteria. Reasons for this are varied but boil down to the idea that the user will typically either find the desired result in the first 10 or 20 items or go back and try the search again, this time with more stringent criteria.

So what happened to the other 990 rows you retrieved as part of that request? Wasted space, wasted CPU cycles, wasted bandwidth.

There are a couple of ways to control how much data gets pulled back as part of the SQL query. One is at the JDBC level, by setting `setFetchSize` to retrieve only a number of items equivalent to the display window you wish to present to the user: if you're showing only the first 10 items retrieved, then use `setFetchSize(10)` on the `ResultSet` before retrieving the first row via `next`. This way, you're certain that only that number of rows is retrieved, and you know that you're making one round-trip to retrieve this data you know you're going to use. Alternatively, you can let the driver try to keep what it believes to be the optimal fetch size for that driver/database combination, and instead choose to limit the absolute number of rows returned from this statement by calling `setMaxRows`; any additional rows beyond the number passed in to `setMaxRows` will be silently dropped and never sent.

Another approach, supported by some databases, is to use the `TOP` qualifier as part of the request itself, as in `SELECT TOP 5 first_name, last_name FROM Person WHERE...`. Only the first five rows that meet the predicate will be returned from the database. Although a nonstandard extension, it's still useful and is supported by a number of the database vendors. `TOP` is SQL Server syntax; other databases use terms like `FIRST` (Informix), `LIMIT` (MySQL), or `SAMPLE` (Oracle). All work similarly.

Again, the idea is simple: only pull back what data is required at this time, on the grounds that you want to avoid pulling back excessive amounts of data that won't be used. Be careful, however; you're standing on a slippery slope when you start thinking about lazy-loading, and if you're not aware of it, it's easy to find yourself in a situation where you're excessively lazy-loading data elements across the wire, resulting in what Fowler

calls "ripple-loading" or as it's more commonly known, the *N*+1 query problem [Fowler, 270]. In those situations, sometimes it's better to eager-load the data (see Item 47) in order to avoid network traffic.

Item 47: Eager-load frequently used data

Eager-loading is the opposite of lazy-loading (see Item 46): rather than taking the additional network round-trip to retrieve data later, when it's needed, we decide to pull extra data across the wire now and just hold it on the database client on the grounds that we'll need it eventually.

This idea has a couple of implications. First, the payoff—actually accessing the extra data—has to justify the cost of marshaling it across the wire from server to client. If the data never gets used, eager-loading it is a waste of network bandwidth and hurts scalability. For a few columns in a single-row retrieval query, this is probably an acceptable loss compared with the cost of making the extra round-trip back to the database for those same columns. For 10,000 rows, each unused column just plain hurts, particularly if more than one client executes this code simultaneously.

Second, we're also implicitly assuming the cost of actually moving the data across the wire isn't all that excessive. For example, if we're talking about pulling a large BLOB column across, make sure that this column will be needed, since most databases aren't particularly optimal about retrieving and sending BLOBs across.

You don't hear much about eager-loading data because to many developers it conjures up some horrific images of early object databases and their propensity to eager-load objects when some "root" object was requested. For example, if a `Company` holds zero to many `Department` objects, and each `Department` holds zero to many `Employee` objects, and each `Employee` holds zero to many `PerformanceReview` objects . . . well, imagine how many objects will be retrieved when asking for a list of all the `Company` objects currently stored in the database. If all we were interested in is a list of `Company` names, all the `Department`, `Employee`, and `PerformanceReview` objects are obviously a huge waste of time and energy to send across the wire.

In fact, eager-loading has nothing to do with object databases (despite the fact that they were blamed for it)—I've worked with object-relational

layers that did exactly the same thing, with exactly the same kind of results. The problem, although "solved" differently, is the same as that for lazy-loading: the underlying plumbing layer just doesn't know which data to pull back when retrieving object state. So, in the case of the eager-loading system, we err on the side of fewer network round-trips and pull it *all* back.

Despite the obvious bandwidth consumption, eager-loading does have a number of advantages to it that lazy-loading can't match.

First, eager-loading the complete set of data you'll need helps tremendously with concurrency situations. Remember, in the lazy-loaded scenario, it was possible (without Session Façade [Alur/Crupi/Malks, 341] in place) for clients to see semantically corrupt data because each entity bean access resulted in a separate SQL query under separate transactions. This meant that another client could modify that row between transactions, thus effectively invalidating everything that had been retrieved before, without us knowing about it. When eager-loading data, we can pull across a complete dump of the row as part of a single transaction, thus obviating the need for another trip to the database and eliminating the corrupt semantic data possibility—there's no "second query" to return changed data. In essence, eager-loading fosters a pass-by-value dynamic, as opposed to lazy-loading's pass-by-reference approach.

This has some powerful implications for lock windows (see Item 29), too. The container won't have to hold a lock against the row (or page, or table, depending on your database's locking defaults) for the entire duration of the session bean's transaction, if accessed via EJB, or the explicit transaction maintained either through a JTA `Transaction` or JDBC `Connection`. Shorter lock windows mean lower contention, which means better scalability.

Of course, eager-loading all the data also means there's a window of opportunity for other clients to modify the data, since you're no longer holding a transactional lock against it, but this can be solved through a variety of means, including explicit transactional locks, pessimistic concurrency models (see Item 34), and optimistic concurrency models (see Item 33).

Second, as already implied, eager-loading data can drastically reduce the total time spent on the wire retrieving data for a given collection if that data can be safely assumed to be needed. Consider, for example, user preferences: data that individual users can define in order to customize their

use of the application in a variety of ways, such as background images for the main window, whether to use the "basic" or "advanced" menus, and so on. This is data that may not be needed immediately at the time the user logs in, but it's a fair bet that all of it will be used at some point or another within the application. We could lazy-load the data, but considering that each data element (configuration item) will probably be needed independently of the others, we're looking at, again, an *N*+1 query problem, in this case retrieving each individual data element rather than individual row. Go ahead and pull the entire set across at once, and just hold it locally for use by the code that needs to consult user preferences when deciding how to render output, windowing decorations, or whatever.

Eager-loading isn't just for pulling back columns in a row, either. As with lazy-loading, you can apply eager-loading principles at scopes larger than just individual rows. For example, we can apply eager-loading principles across tables and dependent data, so that if a user requests a `Person` object, we load all of the associated `Address`, `PerformanceReview`, and other objects associated with this `Person`, even though it might mean multiple queries executed in some kind of single-round-trip batch form (see Item 48).

While we're at it, there's never any reason why a table that holds read-only values shouldn't be eager-loaded. For example, many systems put internationalized text (like days of the week, months of the year, and so on) into tables in the database for easy modification at the client site. Since the likelihood of somebody changing the names of the days of the week is pretty low, go ahead and read the entirety of this table once and hold the results in some kind of in-process collection class or `RowSet` for later consultation. Granted, it might make system startup take a bit longer, but end users will see faster access on each request-response trip. (In many respects, this is just a flavor of Item 4.)

In the end, eager-loading data is just as viable and useful an optimization as lazy-loading data, despite its ugly reputation. In fact, in many cases it's a far more acceptable tradeoff than the lazy-loading scenario, given the relatively cheap cost of additional memory compared with the expensive and slow network access we currently live under. As always, be sure to profile (see Item 10) before doing either lazy- or eager-loading optimizations, but if an eager-load can save you a couple of network accesses, it's generally worth the extra bandwidth on the first trip and the extra memory to hold the eager-loaded data.

Item 48: Batch SQL work to avoid round-trips

Given the cost of moving across the network, we need to minimize the number of times we travel across the network connection to another machine. Unfortunately, the default behavior of the JDBC driver is to work with a one-statement, one-round-trip model: each time `execute` (or one of its variations, `executeUpdate` or `executeQuery`) runs, the driver marshals up the request and sends it to the database where it is parsed, executed, and returned. This tedious process consumes many CPU cycles in pure overhead on each trip. By batching statements together, we can send several SQL statements in a single round-trip to the database, thus avoiding some of that overhead.

Bear in mind that while this is predominantly a state management issue, trying to improve performance by reducing the number of times we have to hit the network, it also applies to transactional processing. We're likely to be in a situation where we're holding open transactional locks, and therefore we want to minimize the amount of time those locks are held open (see Item 29).

There's another element to this, however; sometimes multiple statements need to be executed in a group in order to carry out a logical request. Normally, you would want to do all this under a single transaction (thus requiring that you `setAutoCommit(false)` on the `Connection` you're using), but that doesn't imply that the driver will do it all as part of a single round-trip. It's entirely plausible that the driver will send each statement over individually, all while holding the transactional lock open. Therefore, since it's all happening under a single transaction, it's far better to execute them as a group. The transaction itself is an all-or-nothing situation anyway.

```
Connection conn = getConnectionFromSomeplace();
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();

// Step 1: insert the person
stmt.addBatch("INSERT INTO person (id, f_name, l_name) " +
  "VALUES (" + id + ", '" + firstName + "', '" +
  lastName + "')");
// Step 2: insert the person's account
stmt.addBatch("INSERT INTO account (personID, type) " +
  "VALUES (" + id + ", 'SAVINGS')");
```

```
// Execute the batch
int[] results = stmt.executeBatch();
// Check the batched results
boolean completeSuccess = true;
for (int i=0; i<results.length; i++)
{
  if (results[i] >= 0 || results[i] ==
    Statement.SUCCESS_NO_INFO)
    /* Do nothing—statement was successful */ ;
  else
  {
    /* Something went wrong; alert user? */
    completeSuccess = false;
  }
}

if (completeSuccess)
  conn.commit();
else
  conn.rollback();
```

In this code, we create a `Statement` and use it to execute two non-`SELECT` SQL statements against the database. (Batching doesn't work with `SELECT` statements.) In this case, we're adding a person to the database and adding information about the person's new savings account. Note that the key to executing in batch here is to use the `executeBatch` method on `Statement`; this tells the driver to send the statements over to the database en masse.

The `executeBatch` method returns an array of integers as a way to indicate the success or failure of each of the batched methods. Each element in the array corresponds to one batched statement; a zero or positive number indicates the "update count" result (the number of rows affected by the call), and a value of `SUCCESS_NO_INFO` indicates the call succeeded but didn't have an update count.

In the event one of the statements fails, JDBC allows the driver to take one of several options. It can throw an exception and halt execution of statements at that point, or it can continue processing statements, in which case the integer in the results array will be set to `EXECUTE_FAILED`.

Note that executing in batch doesn't imply transactional boundaries—the two statements have executed, but because `AutoCommit` is turned off, they have not yet committed to the database. Therefore, we need to either commit or roll back the work done; in this case, we commit only if all the statements executed successfully, a reasonable policy. If you call `commit` before calling `executeBatch`, the batched statements will be sent over as if you'd called `executeBatch` just prior to `commit`. If you try to batch with `AutoCommit` set to `true`, the results are undefined by the JDBC Specification—which is shorthand for "quite likely to yield some kind of exception."

Using JDBC's `executeBatch` method isn't the only way to batch execution, however. Because many databases support some kind of logical line termination character, thus creating the ability to execute multiple SQL statements as part of one logical line of input, it's possible to write JDBC calls like this:

```
Connection conn = ...; // Obtain from someplace
Statement stmt = conn.createStatement();
stmt.execute("SELECT first_name, last_name FROM Person " +
    "WHERE primary_key = 1234; " +
    "SELECT ci.content, ci.type " +
    "FROM ContactInfo ci, PerConLookup pc " +
    "WHERE ci.primary_key = pc.contactInfo_key " +
    "AND pc.person_key = 1234");
ResultSet personRS = stmt.getResultSet();
    // The first SELECT

if (stmt.getMoreResults())
{
  // The other SELECT
  ResultSet contactInfoRS = stmt.getResultSet();
}
```

Although more awkward to work with,[5] batching statements this way carries the advantage of working for both `SELECT` statements as well as `INSERT`, `UPDATE`, and `DELETE` statements, all while furthering the basic goal, that of trying to amortize the cost of several SQL queries by running them in a single request-response cycle against the database.

5. In production code you should check the Boolean return value from `execute` to make sure the first `SELECT` statement produced results, rather than ignore it as I have here for clarity.

Item 49: Know your JDBC provider

Despite all the revisions and years behind it, the JDBC Specification still doesn't mandate a large number of particulars—deliberately. For example, when a `ResultSet` is created, typically (although again, this is purely by convention, not a requirement) the `ResultSet` holds only the first *N* rows, where *N* is some number that sounded good to your JDBC vendor. For some of the major vendors, this *N* value is—I'm not kidding you—one. Fortunately, this value is not only discoverable but also configurable via `getFetchSize` and `setFetchSize` on the `ResultSet` API, but the point still remains: Do you know what the default is, and is it acceptable? Be sure to check the Boolean return value from `setFetchSize` to ensure you aren't asking for a fetch size that's larger than what the driver and/or the database supports.

Many features of JDBC aren't available, depending on the capabilities of the driver, and your decision regarding how to build the JDBC code using the driver could be deeply affected based on this information. For example, when retrieving data from the `ResultSet`, there appear to be two entirely equivalent ways to obtain the data: either by ordinal position within the `ResultSet` or by column name. Is there any real difference between them, besides perhaps programmer convenience?

It turns out that there's a very real difference between them, depending entirely on your JDBC driver implementation. The first release of the JDBC Specification (the version that shipped with JDK 1.1) only mandated that JDBC drivers provide *firehose cursor* support—that is, once a row or column has been pulled from the cursor, it can never be obtained again. This has huge implications when retrieving data out of the `ResultSet`, in that if you pull a column's data in anything other than the order in which it was declared in the SQL statement, you'll be skipping past columns that can then never be retrieved:

```
ResultSet rs =
    stmt.executeQuery("SELECT id, first_name, last_name " +
                      "FROM person");
while (rs.next())
{
  String firstName = rs.getString("first_name");
  String lastName = rs.getString("last_name");
  int id = rs.getInt("id");
```

```
    // Error! This will throw a SQLException in a JDBC 1.0
    // driver
}
```

Say you've been working with a JDBC 2.0 or better driver up until this point. If somebody deploys your code to use the JDBC-ODBC driver (which is a bad idea from the beginning, since the JDBC-ODBC driver is an unsupported, bug-ridden 1.0 driver that is incredibly slow and is rumored to leak memory in some ODBC driver configurations), suddenly your code will start tossing `SQLException` instances for no apparent reason.

Now, if you are truly concerned about writing portable J2EE code, you need to account for the possibility of a 1.0 driver by ensuring that this particular scenario never occurs; therefore, you need to always retrieve data in its exact order, and the easiest way to do this is to use the ordinal form of the methods:

```
ResultSet rs =
  Stmt.executeQuery("SELECT id, first_name, last_name " +
                    "FROM person");
while (rs.next())
{
  int age = rs.getInt(1);
  String firstName = rs.getString(2);
  String lastName = rs.getString(3);
}
```

Although not overly painful to write, this code does have several drawbacks. First, if a programmer (or, potentially worse, a database administrator unfamiliar with this little quirk of the JDBC 1.0 API) ever modifies the SQL statement, the corresponding data-retrieval code in the loop will need to be updated to reflect the change in order of columns, or `SQLException` instances will start to get thrown left and right. (By the way, it's true that using the ordinal form of the functions is slightly faster than using the column-based form, but changing your code to use the ordinal form purely for performance reasons is a micro-optimization and should be avoided, particularly since you lose the inherent documentation hint that using the column names give. If you find yourself tempted to do so, a large number of other optimizations will likely yield a better return, so bypass this one.)

It's not just a simple matter of keeping track of the order of columns (which, by the way, argues against ever issuing a SQL `SELECT` statement that uses `*` instead of the individual column names). You need to know whether your driver supports statement batching (see Item 48) and isolation levels (see Item 35), which scalar functions it might support, and so on. Much of this information is available via the `DatabaseMetaData` class, an instance of which is obtained from a `Connection`, or to a lesser degree from the `ResultSetMetaData` class, obtained from a given `ResultSet`. *SQL Performance Tuning* [Gulutzan/Pelzer] provides several charts describing the results returned from eight vendor databases (IBM, Informix, Ingres, InterBase, Microsoft SQL Server, MySQL, Oracle, and Sybase), but you'll want to run some tests regardless—you can't be certain the chart will be true for your particular database, even if it's one of the vendors listed, because capabilities can change from one release to the next.

Another area of interest to the JDBC programmer is that of thread safety: Is it safe to invoke the driver from multiple threads? Do you need to synchronize on `Connection`, `Statement`, or `ResultSet` objects when accessing them from multiple threads? For example, the JDBC-ODBC driver isn't thread safe, so it's up to you to ensure that the driver is never accessed by more than one thread at a time; otherwise you run the risk of some Seriously Bad Things happening in the ODBC driver underneath your Java code. (Which once again underscores that there's never a good reason to use the JDBC-ODBC driver for production code.)

One of the classic performance- and scalability-driven suggestions made in numerous JDBC and J2EE books is the `PreparedStatement`-over-`Statement` idea: that you should always prefer to use a `PreparedStatement` instead of a regular `Statement`. The argument is simple: because each call out to the database requires a certain amount of work (parsing the SQL, creating a query plan, and running it all through the optimizer, and so forth), it's better to amortize that overhead by keeping those preparations around from one call to the next, assuming the same call will be made again. A `PreparedStatement` will "prepare" the SQL call once (minus the parameters you want to pass in, since those aren't known yet, so it can't prepare those) and thus you'll get better performance on subsequent calls.

While using `PreparedStatement` is a necessity from a security standpoint (see Item 61), from a performance and/or scalability standpoint, it's not quite so clear. For example, the JDBC Specification states that when a

`Connection` is closed, the corresponding `Statement` objects associated with that `Connection` also implicitly close. So when you return a `Connection` obtained from a connection pool back to the connection pool, does that in turn close the `PreparedStatement` obtained from that `Connection`, or will the underlying physical connection remain open, thus leaving the `PreparedStatement` alive and ready to receive additional requests? What about a `CallableStatement`, since it inherits from `PreparedStatement` and therefore should obey the same contract as given for `PreparedStatement`?

Unfortunately, while I could wave my hands here and state that "obviously, REAL drivers would keep the pooled `Statement` around, only a simpleton or moron wouldn't do this," the fact remains that sometimes you have to use software written by simpletons and morons, and the only way to know for certain whether your database and/or database driver is in that category is to test and find out.

This discussion of `PreparedStatement` takes a sharp left turn, by the way, when discussing some database vendors (most notably PostgreSQL) that cause a `PreparedStatement` to become unprepared when a transaction end (`COMMIT`) occurs; that is, the following code [Gulutzan/Pelzer, 336] won't work as expected, despite the JDBC Specification's insistence that it should:

```
PreparedStatement pstmt =
    connection.prepareStatement(". . .");
boolean autocommit =
    connection.getAutoCommit(); // Let's assume it's true
pstmt.executeUpdate();
    // COMMIT happens automatically, since autocommit == true
pstmt.executeUpdate();
    // FAILS! PreparedStatement pstmt is no longer prepared
```

If that doesn't convince you that you need to know what your JDBC driver does under the hood, then I wish you the best of luck.

The easiest way, in many respects, to know what your driver does is to fire up your database's query-execution profiler tool and have a look at what actually happens when the JDBC query executes. In some circles, this is known as *exploration testing* and is more extensively documented by Stu Halloway (http://www.relevancellc.com); put simply, write a series of JUnit-based tests that exercise your assumptions regarding what happens

when you execute a sequence of calls (such as the `PreparedStatement` code shown earlier), and rerun those tests against a variety of scenarios, such as different JDBC drivers, different vendor databases, or even different versions of the same vendor's database. Far better to be surprised during an exploration test run than to be surprised during deployment into production, at the client site, or in the middle of the night after the database administrators upgrade the database without your knowledge (or consent).

Item 50: Tune your SQL

Quiz time: When are two logically equivalent SQL statements not equivalent? Consider the following pair of logically equal SQL statements:

```
SELECT * FROM Table WHERE column1='A' AND column2='B'

SELECT * FROM Table WHERE column2='B' AND column1='A'
```

Which would you say executes faster?[6]

Answer: You can't tell, and neither can the optimizer in most databases. But if you happen to know, for your particular database schema, this particular client, and/or this particular query, that the likelihood of `column2` being `B` is a lot less likely, then the second SQL statement will execute faster than the first, even though the two are logically equivalent. (By the way, never do this for Oracle databases when using the cost-based optimizer—it will do exactly the wrong thing.)

Or, as another example, how about these two statements?

```
SELECT * FROM Table WHERE column1=5 AND
    NOT (column3=7 OR column1=column2)

SELECT * FROM Table WHERE column1=5 AND column3<>7 AND
    column2<>5
```

Answer: For five of eight popular databases, the second turns out to be faster.[7] Again, these are logically identical statements, so the actual results

6. This example appears in *SQL Performance Tuning* [Gulutzan/Pelzer, 24].

7. This example appears in *SQL Performance Tuning* [Gulutzan/Pelzer, 16].

returned will be the same; it's just that the database optimizer treats the second one in a different fashion than the first, thereby yielding a better response time.

Tuning SQL remains the number one optimization you can make when working with a relational database. Despite what JDO and entity bean vendors have been trying to achieve for the last five years, we're still in an era where SQL matters. Yes, vendors have made huge strides in making their objects-first persistence models not suck as badly as they first did, but the fact remains that if the database itself can't tell the difference between the two statements, it's highly unlikely that your object-relational layer will be able to. This is why it's crucial for an object-relational system to offer you, at the least, some kind of hook point (see Item 6) to pass in raw SQL that you can tune and optimize as necessary for those queries executed most often.

By the way, bear in mind that for some databases, these two statements are considered to be entirely different and therefore require reparsing, replanning, and reexecution:

```
SELECT column1*4 FROM Table1 WHERE COLUMN1=COLUMN2 + 7

SELECT Column1 * 4 FROM Table1  WHERE column1=(column2 + 7)
```

That is to say, even though these statements are precisely identical in what they're doing, because the capitalization and whitespace used within them are different, the database treats them as separate and unrelated statements. With a tool generating your queries, we would hope that they're being generated in a consistent style, but can you be certain? (Ideally, you'd know whether this sort of thing bothers your database before worrying about it too much, but that means you're OK with *a priori* vendor awareness like that; see Item 11.)

In many cases, you need to know what SQL is actually being executed against your database before you can think about trying to tune it. For some object-relational systems this can be a difficult prospect if they don't expose the actual generated or dynamically constructed SQL code to you. Fortunately, an answer is only a short step away.

First, most database products offer some kind of database-based view of queries being executed against a given database instance, so it's usually pretty easy to fire up the database profiler tool and take a look at the actual SQL. (While you're at it, assuming your database provides query

analysis as part of that same profiler, take the SQL queries and run them against the profiler to see how the database will attack executing this particular query; see Item 10.) This should tell you how your favorite object-relational tool is generating the SQL queries and thereby give you a good idea whether SQL optimization is in order.

If you're working with a database that doesn't provide such a tool (time to get a new database, in my opinion, but sometimes you have to work with the tools at hand), assuming your object-relational layer is still going through JDBC to talk to the database itself, you can play a small trick on the object-relational layer by handing it a JDBC driver that "leaks" the actual queries being fed through it. It's the P6Spy driver, available from http://www.p6spy.com, and its operation is deceptively simple: it exposes the same JDBC driver interface that every other JDBC driver on the planet provides, but it doesn't do any actual work, instead delegating that to a JDBC driver for which you provide configuration data. Instead, the P6Spy driver simply echoes the SQL queries to multiple sources, including Log4J logging streams.

For those situations where your object-relational layer isn't doing the optimal SQL thing and the object-relational layer doesn't provide some kind of hook point to pass in optimal SQL, you have a few options.

- *Switch tools.* There are plenty of other object-relational tools out there, both commercial and open-source, that you shouldn't feel "stuck" with one that doesn't give you the necessary optimization hook points.
- *Follow the Fast Lane pattern* [Alur/Crupi/Malks, 1st ed.], which advocates the idea of doing direct JDBC access against the database, thereby giving you direct control over the SQL being sent. In the case of EJB-based access, this means either doing direct JDBC calls from your session beans or rewriting your CMP entity beans as BMP beans. Unfortunately, this pattern has a number of direct drawbacks (which is probably why it's not present in the second edition), including the fact that if the object-relational layer is doing any sort of cache management, you'll be bypassing that cache management and any implicit Identity Map [Fowler, 195] that's being maintained. Perhaps you can live with the loss of the caching behavior; however, if the Identity Map [Fowler, 195] is being maintained in order to allow the object-relational layer to hold off sending any uncommitted changes to the data, you'll bypass that and create a nasty little concurrency problem for yourself.

- *Pull a bait-and-switch with your object-relational tool,* taking a cue from the P6Spy driver (a classic Decorator [GOF, 175] if ever there was one). Write a JDBC driver look-alike—that is, a class that implements the JDBC driver interfaces—that listens for particular queries that match the queries you want to optimize, and replace those queries with hand-tuned, optimized SQL before handing the query off to the real JDBC driver you've wrapped around. You do take an extra layer of method calls as a hit to the driver, but assuming you've done your homework (see Item 10), the gains from the tuned SQL will more than offset the cost of the additional method call.

Whichever method you choose, before you start caching off JNDI lookup results as an optimization, take a hard look at the SQL being executing on your behalf—more often than not, optimizations in the SQL can yield a far greater return on your investment in time and energy than anything else you attempt. Even if your CMP entity bean layer is doing some kind of adaptive SQL generation scheme, you'll want to know about it and make sure that your database administrator is doing his or her part not to work against that.