Course Announcements

- Bacon is due next Monday.
  - Today’s lecture will help thinking about your DB interface.
- Next lab is about *drawing* UIs.
ORMs

John Jannotti

/course/cs0320/www/lectures/

Mar 9, 2017
Rows as Objects

- To a rough approximation, rows can be mapped to objects.
  - table ⇔ class
  - row ⇔ object
  - column ⇔ primitive field/attribute
- An “ORM” is a software layer to make that mapping easy.
  - Object-Relational Mapping.
- But there’s a lot more to it.
  - And it should be said, some argue against ORMs.
  - Learn more: “object relational impedance mismatch”.
- You won’t use an ORM in Bacon, but you should think about abstraction.
Relationships Between Objects

- table ⇔ class
- row ⇔ object
- column ⇔ primitive field/attribute
- ? ⇔ object references

An employee table

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim Jones</td>
<td>98293</td>
<td>?</td>
</tr>
<tr>
<td>Steve Johnson</td>
<td>87293</td>
<td>?</td>
</tr>
<tr>
<td>Greg Smith</td>
<td>129349</td>
<td>?</td>
</tr>
</tbody>
</table>
### Primary Keys

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Symbol</th>
<th>Price</th>
<th>Employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Microsoft</td>
<td>MSFT</td>
<td>30.85</td>
<td>35010</td>
</tr>
<tr>
<td>2</td>
<td>Google</td>
<td>GOOG</td>
<td>595.03</td>
<td>20100</td>
</tr>
<tr>
<td>3</td>
<td>Oracle</td>
<td>ORCL</td>
<td>26.22</td>
<td>18350</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Salary</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>234</td>
<td>Jim Jones</td>
<td>98293</td>
<td>2</td>
</tr>
<tr>
<td>129</td>
<td>Steve Johnson</td>
<td>87293</td>
<td>3</td>
</tr>
<tr>
<td>233</td>
<td>Greg Smith</td>
<td>129349</td>
<td>2</td>
</tr>
</tbody>
</table>
Difficulties with Collections

- A row may not contain an arbitrary number of anything.
- To encode a many-to-one relationship between employees and companies:
  - A company ID is embedded in each employee row.
  - May feel “backwards”
  - This is a “foreign key” and there are consistency checks.
- Foreign keys provide associated sets, but not lists, and not many-many relations.
### Employees with multiple companies

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Symbol</th>
<th>Price</th>
<th>Employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Microsoft</td>
<td>MSFT</td>
<td>30.85</td>
<td>35010</td>
</tr>
<tr>
<td>2</td>
<td>Google</td>
<td>GOOG</td>
<td>595.03</td>
<td>20100</td>
</tr>
<tr>
<td>3</td>
<td>Oracle</td>
<td>ORCL</td>
<td>26.22</td>
<td>18350</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>234</td>
<td>Jim Jones</td>
<td>98293</td>
</tr>
<tr>
<td>129</td>
<td>Steve Johnson</td>
<td>87293</td>
</tr>
<tr>
<td>233</td>
<td>Greg Smith</td>
<td>129349</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Employee ID</th>
<th>Company ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>234</td>
<td>2</td>
</tr>
<tr>
<td>129</td>
<td>3</td>
</tr>
<tr>
<td>233</td>
<td>2</td>
</tr>
<tr>
<td>129</td>
<td>1</td>
</tr>
<tr>
<td>233</td>
<td>3</td>
</tr>
</tbody>
</table>
Maybe the relationship itself deserves “promotion” to a type.

Does this relationship deserve its own abstraction? “Employment”
  ▶ Then it has one-to-many relationships to employees and companies.
  ▶ More importantly, a natural place for associated data.
  ▶ For example, salary belongs there, not in employee.

Same schema, except the intermediate table is now first-class.
  ▶ And now you have two one-to-many relationships, instead of one
    many-to-many.

How might we take this advice in the Bacon schema?
Advice for database-backed applications

- Identify your “units of work” (often a single web, or API request)
- Use transactions to isolate those units.
- You can abort cleanly because of atomicity.
- Keep your state in the database (not in memory).
  - Scaling is more straightforward.
  - A (single!) caching layer makes it fast enough.
  - Large binary data (e.g. images) may be an exception.
We’ve talked about the schema you might use to store objects. How would you structure the code itself? Think about:

- Separating “business logic” from “database access”.
- Creating new objects (allocating IDs, explicit saves)
- Exposing/Hiding the amount of “laziness” for fetching object graphs.
- Maintaining object identity (programmer expectations, efficiency).
- Performance: Caching lookups, managing connections, concurrency.
- Sharing code across different objects, collections.
What do you want an ORM to do for you?

- **CRUD: Create, Read, Update, Delete**
  - These are the basics. You want to do these without SQL.
  - Simple objects are simple, what about references?
  - Should the ORM “chase pointers”? Monitor “dirtiness”?
  - Does the client write explicit operations?

- **Caching**
  - As you traverse your “business objects”, you may cycle.
  - It would be bad to keep querying the DB.
  - It would also be bad to have multiple “Toy Story” objects.

- **Declarative style**
  - Mapping to the existing schema.
  - How much to prefetch vs be lazy.
  - Maybe a config file, maybe attributes.
Clicker Test

Which are you struggling with most?
A) Opening, passing, sharing, closing DB resources.
B) Performance (Including, but not limited to caching)
C) Writing the SQL queries. Keeping them out of “business logic”
D) Unit testing Dijkstra (while avoiding the database).
E) Making your graph generic across node, edge types.
One way to do it

- Let’s look at a simpler way you might do this for Bacon.
- I tell you explicitly: I don’t think this is best, but it’s decent.
- That said, don’t try to implement Hibernate, stay away from Reflection APIs.
- First, two sidebars, to understand some of the code
  - try-with-resources: a much better way to close objects that need closing.
  - lambda expression: concise syntactic sugar for anonymous classes (even anonymous classes may be new to you).
Try With Resources

- You used to think Garbage Collection would always clean up.
- You found that wasn’t true for open files.
- And lately, database connections, statements, results, etc.
- Until Java 7, handling these things properly was challenging.

```java
public void badCode() {
    InputStream stream = new FileInputStream(...);
    // ... use stream
    stream.close();
}
```

This code can’t be counted on to close() the stream properly.
“finally” helps, sort of

```java
public void tediousCode() {
    InputStream stream;
    try {
        stream = new FileInputStream(...);
        // ... use stream
    } catch (IOException e) {
        // handle it
    }
    finally {
        try {
            if (stream != null) {
                stream.close();
            }
        } catch (IOException e) {
            // handle yet another one
        }
    }
}
```
try-with-resources to the rescue

```java
public void cleanCode() {
    try (InputStream stream = new FileInputStream(...)) {
        // ... use stream
    } catch (IOException e) {
        // handle it
    }
}
```
What are lambda expressions (in Java)?

- A concise expression to represent a function.
- But no changes at the JVM level.
- So what must a lambda expression **really** create?
Representing functions before lambda

- The Comparator interface allows Collections.sort() use the caller’s notion of comparison.
- Think of an object (that implements Comparator) as a function.
- To make a new one, you need a new class (AgeComparator).

```java
public class AgeComparator
    implements Comparator<Person>
{
    public int compare(Person x, Person y)
    {
        return x.getLastName().compareTo(y.getLastName());
    }
}
Collections.sort(people, new AgeComparator());
```
Anonymous class creation is verbose

Anonymous classes make it a bit easier, but...

```java
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

Both the meaning under boilerplate.
Java “Lambdas” are syntactic sugar

```java
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

becomes

```java
Collections.sort(people, (x, y) -> x.getLastName().compareTo(y.getLastName()));
```
What does a lambda do?

- *Exactly* the same thing as the previous code.
- How? We didn’t say anything about Comparator.
What does a lambda do?

- *Exactly* the same thing as the previous code.
- How? We didn’t say anything about Comparator
- Collections.sort “wants” a Comparator
- Comparator is a *Functional interface* (one non-default method)
- So that’s what the lambda creates.
- (If sort were overloaded, you might need to disambiguate.)
- Important: There’s nothing *really* new here. Just convenience.
- “Method reference” syntax, Person::getName, also instantiates a functional interface.
On to example DB code.

Highlights to look for

- The concept of an Entity (rather than a Value class)
- The use of the Proxy pattern (to hit the DB lazily)
- The use of Actor and Movie interfaces (to allow Proxy and Bean implementations).
- The “purity” of the Bean implementation. (No SQL anywhere.)
- The caching layer. All in one place, EntityProxy.
- The strange but important equals() overrides.