Data Knowledge Objects

By Derek Anderson

BIO

Co-founder MyDocsDocs.com
Computational Finance
USAF / DOD
MS - CS
BS - Math

http://vosterra.com
derek@vosterra.com

Why Another ORM?

- Most ORMs are built on a questionable assumption:
  - Database access code is hard to {write, debug, maintain, verify}.

- True for many ORMs because SQL is represented as Strings.
- Still true for ORMs that have query builders that use String identifiers.

Consequences

- Database code isolation!
  It's a nice theory…
- Most code still compile-time checked
- "Dangerous" DB code isolated in its own package

Day #1 of Project

- Methods in “com.company.project.dbcode”:

getInstrument(int id);
getAllInstruments();

Not so bad…

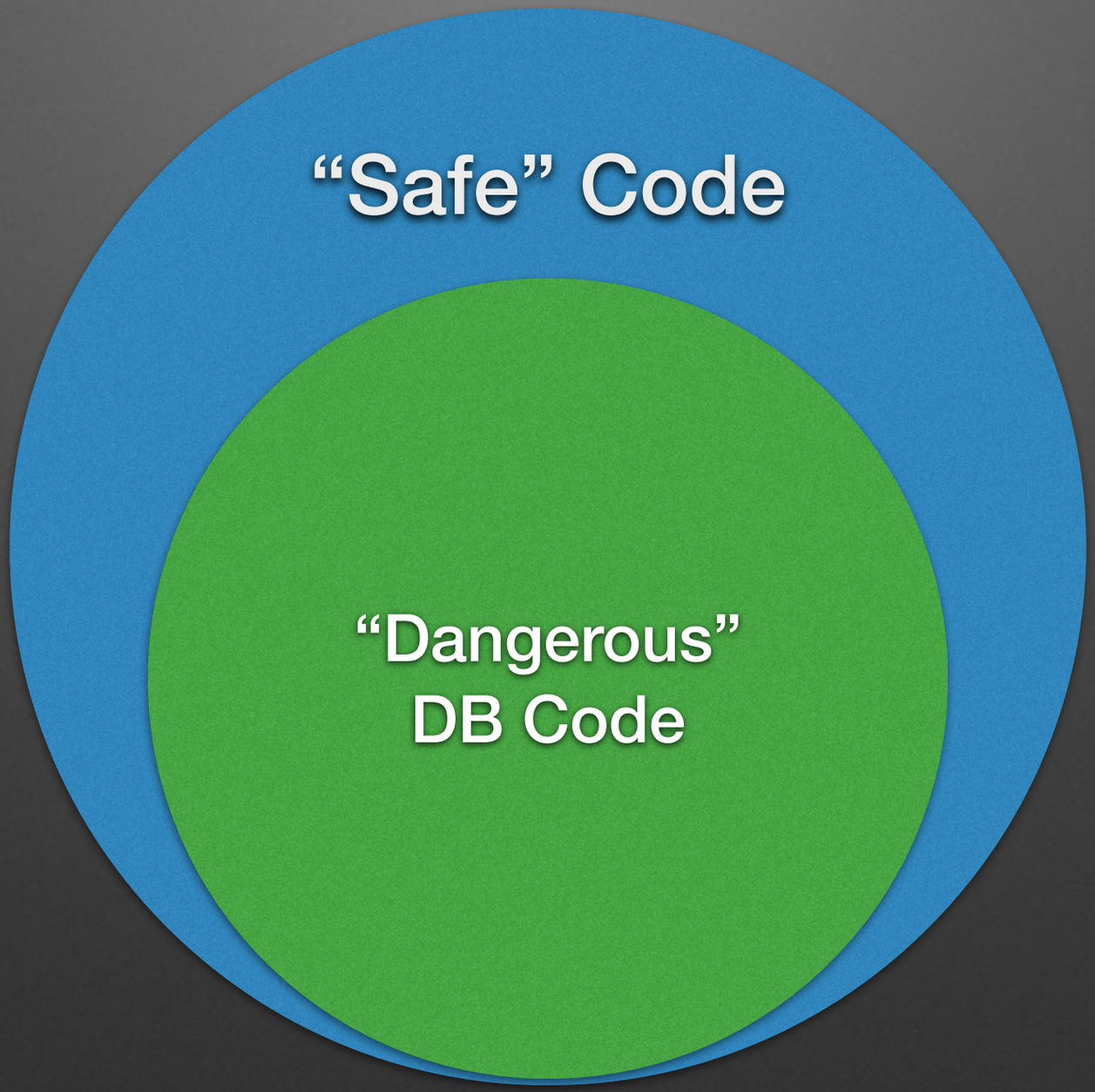
Day #100 of Project

- Methods in “com.company.project.dbcode”:

```
getInstrument(int id);
getAllInstruments();
getInstrumentByName(String name);
getAliveInstruments();
getInstrumentsInMarket(Market m);
getInstrumentByType(InstrumentType t);
getInstrumentsById(int[] ids); // WARNING: <256!
getInstrumentsById(List<Integer> ids);
getInstrumentsAliveBetweenDates(Date d1, Date d2);
getInstrumentsInRegionAndOfType(Region r, InstrumentType type);
// and on and on and on…
```

Not so good…

Reality Sinks In…



Or Even Worse…

- Many (all?) of your programmers do this:

```
Collection<Instrument> instrs = getAllInstruments();
for (Instrument instr : instrs) {
    if( !iWant(instr) ) continue;
    // do the real work
}
```

And Other Problems…

- Do you love XML? Because I have > 35,000 LOC of configuration files right here for ya!
- The terrible idea of a “living database object”.
- Who doesn’t like “SELECT * FROM” all the time?
- Streaming? Come on, who needs that?!?

Clarifying the Assumption

- Assumption v2.0:

- String-based* database access code is hard to {write, debug, maintain, verify}.

  (*) Even String identifier based query builders!

So Lets Simplify

- Embrace the SQL!
  - String manipulation / identifiers are the problem, not database theory.
    So make everything compile-time checked.
    - Of course, this means code generation, but that horror can be mitigated.
- Keep all object<->table mappings 1-to-1!
  - Huge, instantaneous reduction in configuration.
  - Complicated mappings can all be done in Java on top.

Prior Work

- Apache Torque
  http://db.apache.org/torque/
  Sadly no longer maintained...
  (Also it was never updated for generics, and had some bad scalability bugs last time I used it.)

- The Django Query API
  https://docs.djangoproject.com/en/1.4/topics/db/queries/
  Part of a widely used Python framework.
  Largely the inspiration for DKOs.

Hello World

- ```
  Query<Instrument> instrs = Instrument.ALL;
  for (Instrument instr : instrs) {
      // do your work
  }
  ```
- Immutable, iterable “Query” objects centered around the “ALL” constant. (one per table)
- SQL only executed when iteration starts.
- Streaming by default!
  ```

Basic Concepts

- A table: com.company.dko.Instrument.class
- A row: new com.company.dko.Instrument()
- A Field: com.company.dko.Instrument.ID
  Or {NAME, START_DATE, REGION}
- Conditions:
  Instrument.ID.eq(658)
  Instrument.NAME.like(“IBM”)
  Instrument.ID.between(d1, d2)

More Examples

```
// asserts only one record matches
Instrument i = Instrument.ALL
      .get(Instrument.ID.eq(658));

// multiple where clauses are ANDed together
long count = Instrument.ALL
      .where(<Condition...>)
      .where(<Condition...>)
      .count();  // count happens on database

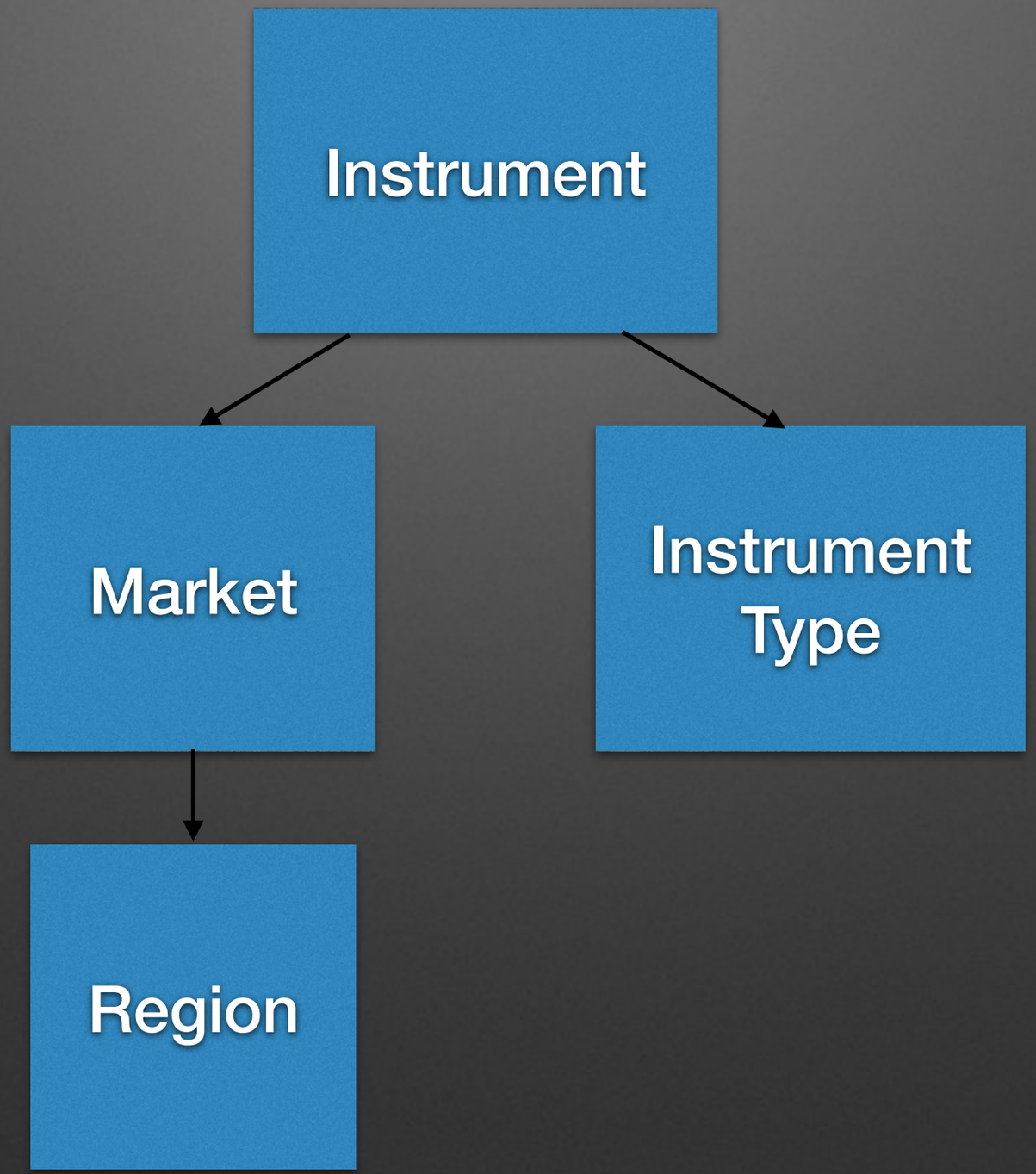
// an example OR statement
Instrument.ID.eq(658).or(Instrument.ID.eq(659))

// an example IN statement
Instrument.ID.in(658, 659, 660, 661)
```

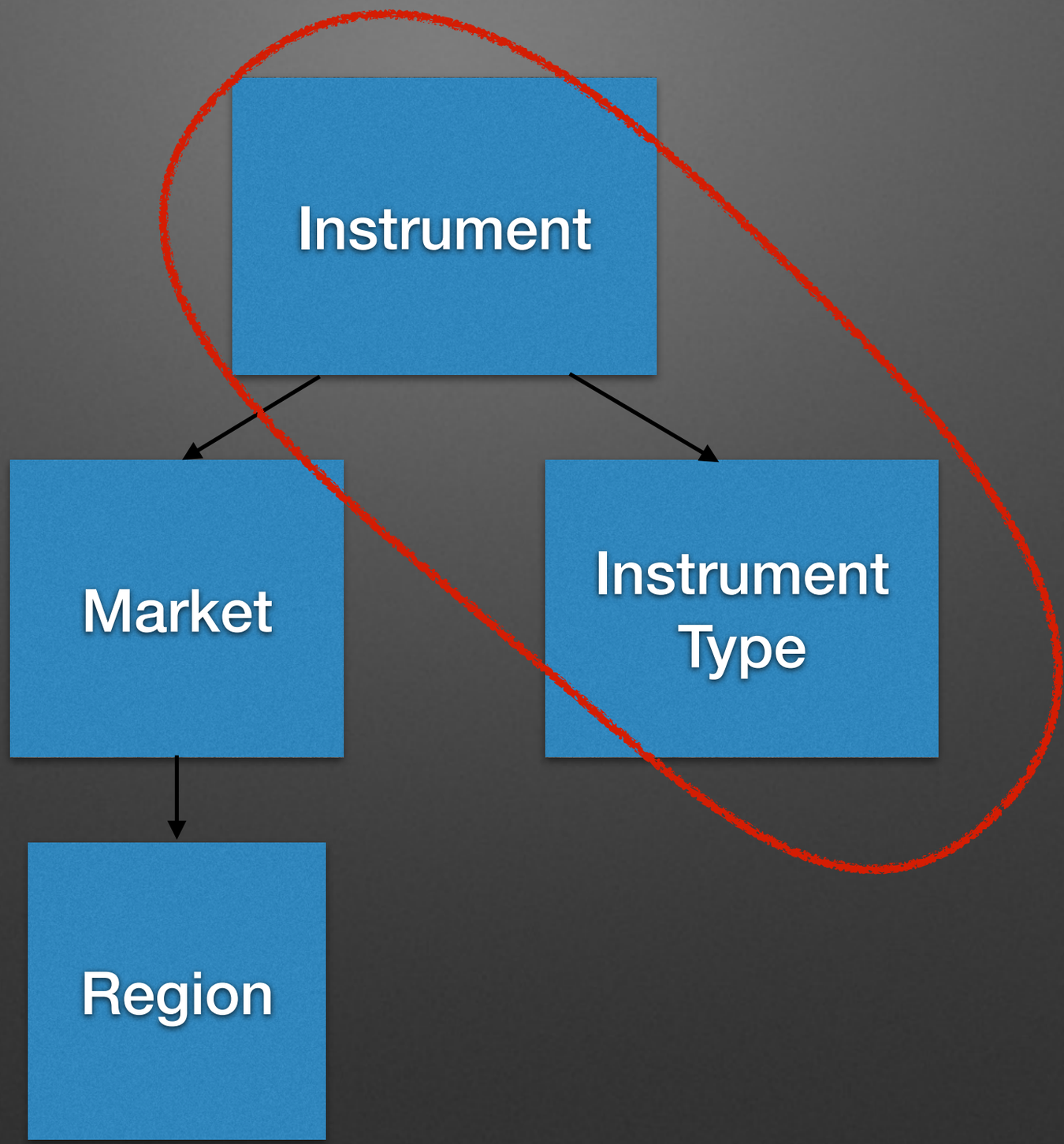
More Supported SQL

- On Queries: count(); orderBy(...); top(int); distinct(); max(); min(); latest(); first(); exists(); sum(...); sumBy(...); countBy(...); mapBy(...); …
- On Fields: eq(); gt(); lt(); add(); mod(); isNull(); in(); …
- Functions: IFNULL(); COALESCE(); GETDATE(); DATEADD(); DATEPART(); DATEDIFF(); CONCAT(); …

Let’s Talk Joins



Let's Talk Joins



Let's Talk Joins

Table "INSTRUMENT" has a foreign key to "INSTRUMENT_TYPE".

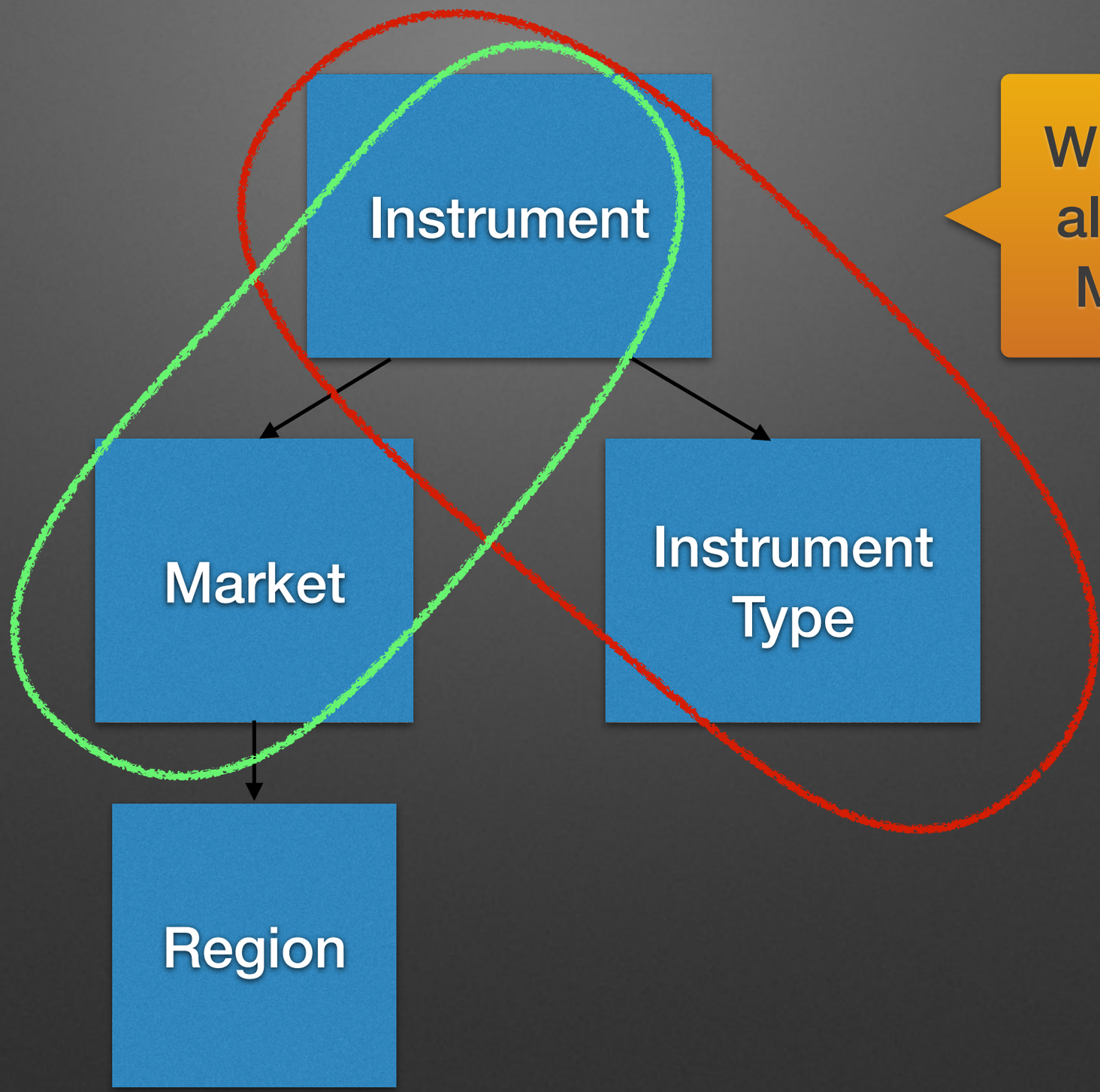
So class com.company.dko.Instrument has a static constant "FK_INSTRUMENT_TYPE".

```
// example join (instrument to instrument_type)
Query<Instrument> q = Instrument.ALL
    .with(Instrument.FK_INSTRUMENT_TYPE);
for (Instrument inst : q) {
    // *NOT* lazy loaded - part of the original joined query
    InstrumentType type = inst.getInstrumentTypeFK();
}
```

Let’s Talk Joins

```
// the other way (instrument_type to instrument)
Query<InstrumentType> q = InstrumentType.ALL
    .with(Instrument.FK_INSTRUMENT_TYPE);
for (InstrumentType type : q) {
    System.out.println(“all: ”+ type);
    // *NO* second query here…
    for (Instrument inst : type.getInstrumentSet()) {
        System.out.println(“\t”+ inst);
    }
}
```

Let's Talk Joins



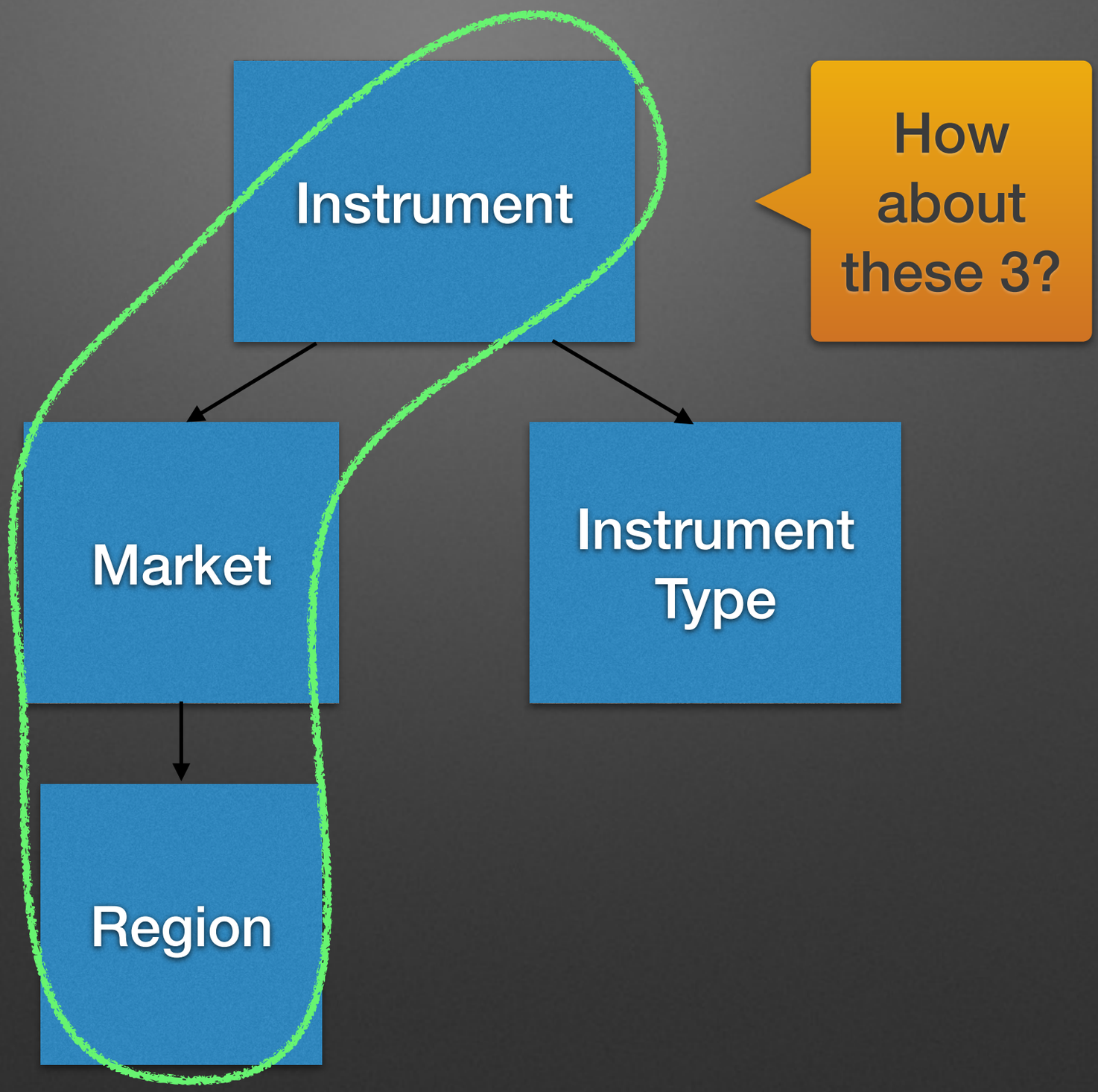
What if we also want Market?

Let's Talk Joins

Joining both InstrumentType and Market:

```
// a 3-way join (only one query)
Query<Instrument> q = Instrument.ALL
    .with(Instrument.FK_MARKET)
    .with(Instrument.FK_INSTRUMENT_TYPE);
for (Instrument inst : q) {
    InstrumentType type = inst.getInstrumentTypeFK();
    Market market = inst.getMarketFK();
}
```

Let's Talk Joins

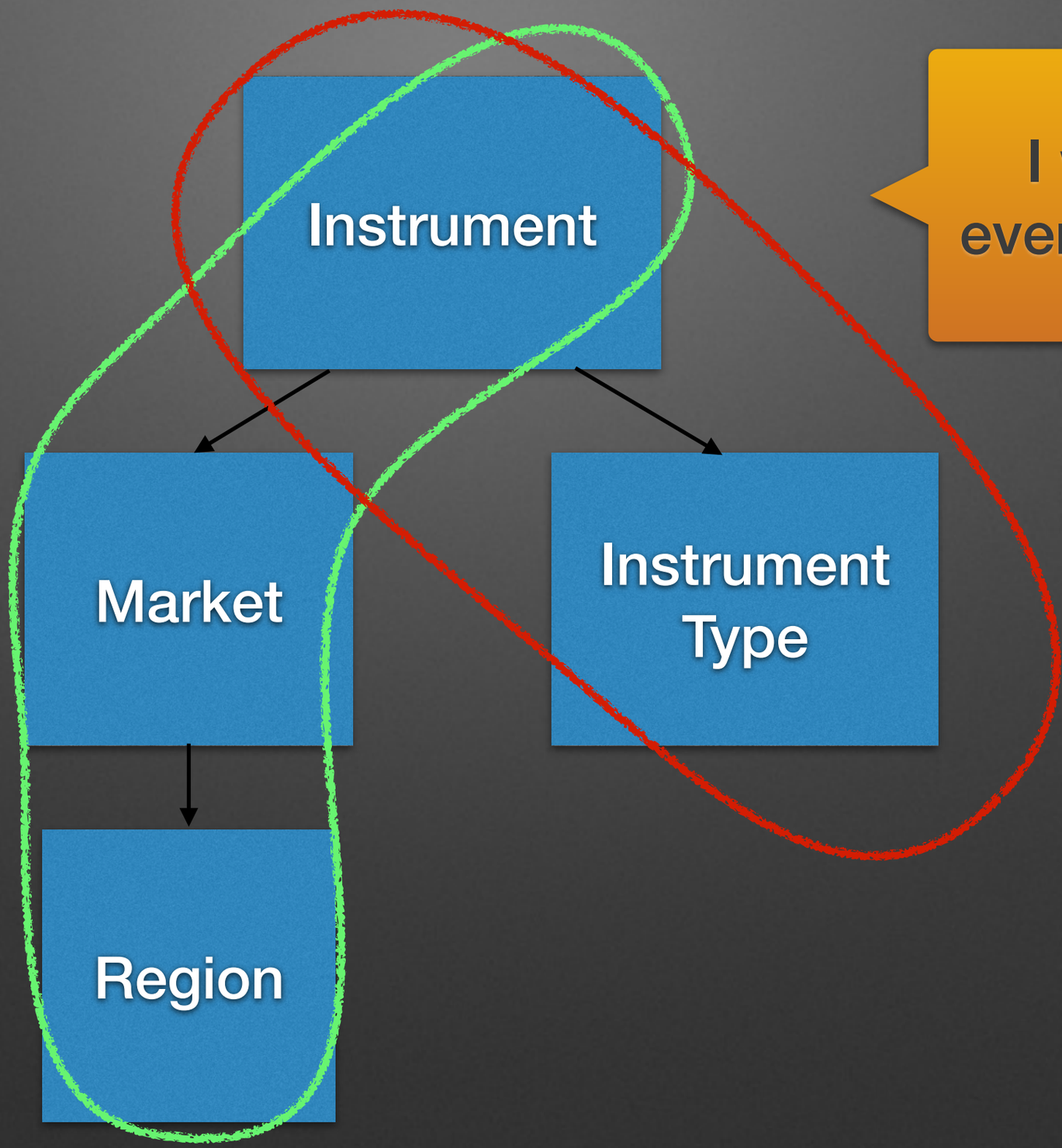


Let's Talk Joins

Joining to Region through Market:

```
// a 3-way join (two levels deep)
Query<Instrument> q = Instrument.ALL
    .with(Instrument.FK_MARKET, Market.FK_REGION);
for (Instrument inst : q) {
    Market market = inst.getMarketFK();
    Region region = market.getRegionFK();
}
```

Let's Talk Joins



Let's Talk Joins

Joining to Region through Market:

```
// a 3-way join (two levels deep)
Query<Instrument> q = Instrument.ALL
    .with(Instrument.FK_INSTRUMENT_TYPE)
    .with(Instrument.FK_MARKET, Market.FK_REGION);
for (Instrument inst : q) {
    InstrumentType type = inst.getInstrumentTypeFK();
    Market market = inst.getMarketFK();
    Region region = market.getRegionFK();
}
```

Inner Queries

This:

```
Query<Instrument> x = Instrument.ALL.where(
      Instrument.ID.in(
            InstrumentInUniverse.ALL.where(
                  InstrumentInUniverse.UNIVERSE_ID.eq(1234)
                        .onlyFields(InstrumentInUniverse.INSTRUMENT_ID)
            ) // the inner query
      ) // a condition
);
```

Becomes...

```
select * from instrument where id in (
      select instrument_id from instrument_in_universe
      where universe_id = 1234
)
```

Temporary Tables

Some databases (*cough* SQLServer *cough*) can't handle large "select where X in ()" statements. So DKOs automatically convert them to joins with temp tables if over a threshold.

```
Set<Integer> ids = getLotsOfInstrumentIds();
System.out.println(ids.size()); // ~20,000!
for (Instrument instr : Instrument.ALL
      .where(Instrument.ID.in(ids))) {
     // this doesn't crash!
}
```

SqlServer otherwise would slow to a crawl at ~ 500 (and crash at > 2048) otherwise.

Math and SQL Functions

Looking for large stock jumps?

```
Price.ALL.where(
    Price.CLOSE_PRICE.gte( // greater than or equal
        Price.OPEN_PRICE.mul(1.5)
    )
);
// add(), sub(), mul(), div(), mod() all also supported
```

Instruments that ended in a February?

```
Instrument.ALL.where(
    DATEPART(Instrument.END_DATE, MONTH).eq(2)
); // many other standard SQL functions
```

Object CRUD

Updates only update the fields you've changed:

```
Instrument instr = [...];
instr.setCurrency(“EUR”);
instr.update();
-- only updates the changed value, not the entire object
update instrument set currency='EUR' where id=658;
```

Deletes use the primary key of the table:

```
Instrument instr = [...];
instr.delete();
```

New rows can be created w/ normal constructors:

```
Instrument instr = new Instrument();
// set some values
instr.insert();
```

Query CRUD

All Query objects have full CRUD methods.

To “kill” all currently alive instruments...

```
int updated = Instrument.ALL.where(
        Instrument.END_DATE.isNull())
    .set(Instrument.END_DATE, NOW())
    .update();
```

Or you can delete “bad” rows...

```
int deleted = Instrument.ALL.where(
        Instrument.START_DATE.isNull())
    .deleteAll();
```

Bulk Operations on Objects

A class for bulk operations on arbitrary sets:

```
Bulk bulk = new Bulk(ds);
bulk.insertAll(<Instruments...>);
bulk.updateAll(<Instruments...>);
bulk.deleteAll(<Instruments...>);
bulk.insertOrUpdateAll(...);
```

All bulk operations use JDBC batching APIs.
All methods are still streaming (they accept Iterable).

Transactions

Uses the “Context” class, available at:

```
Context.getVMContext();
Context.getThreadGroupContext();
Context.getThreadContext();
```

A typical transaction:

```
javax.sql.DataSource ds = Instrument.ALL.getDataSource();
context.startTransaction(ds);
// do whatever here, including calling other methods which
// might do their own database work inside this transaction
context.commitTransaction(ds);
```

Transactions effect all SQL operations in the given context (including what happens in intervening method calls), the scope of which is defined by which Contect object you use.

Query Optimizations

WAKE UP! DON’T MISS THIS!

Query Optimizations

DKOs automatically learn what fields you access:

```
for (Instrument i : Instrument.ALL) {
    System.out.println(i.getName());
}
```

Run #1 will “select *” from the table..... and collect usage statistics.
Run #2 will only run: “select id, name” because it knows how your code uses the returned objects!

Optimizations persist across Java virtual machine instances (using a SQLITE database in your home directory).

Also fields can be manually specified:

```
.onlyFields(Instrument.ID, Instrument.NAME)
```

Lazy Loading Warnings

The usage monitor (optionally) gives hints/warnings/suggestions about too much lazy loading calls (written to the logger). For example:

> Jul 21, 2012 5:34:12 PM org.dko.UsageMonitor warnBadFKUsage
>
> WARNING: This code has lazily accessed a foreign key relationship 100% of the time. This caused 29 more queries to the database than necessary. You should consider adding .with(Item.FK_SUPPLIER) to your join.
>
> This happened at:
> test.db.SharedDBTests.testFKNoWith(SharedDBTests.java:101)
>
> while iterating over a query created here:
> test.db.SharedDBTests.testFKNoWith(SharedDBTests.java:99)
>
> To turn these warnings off, call:
> Context.getThreadContext().enableUsageWarnings(false);

Code Generation

EVIL!!!

Well, OK, how evil? And when?

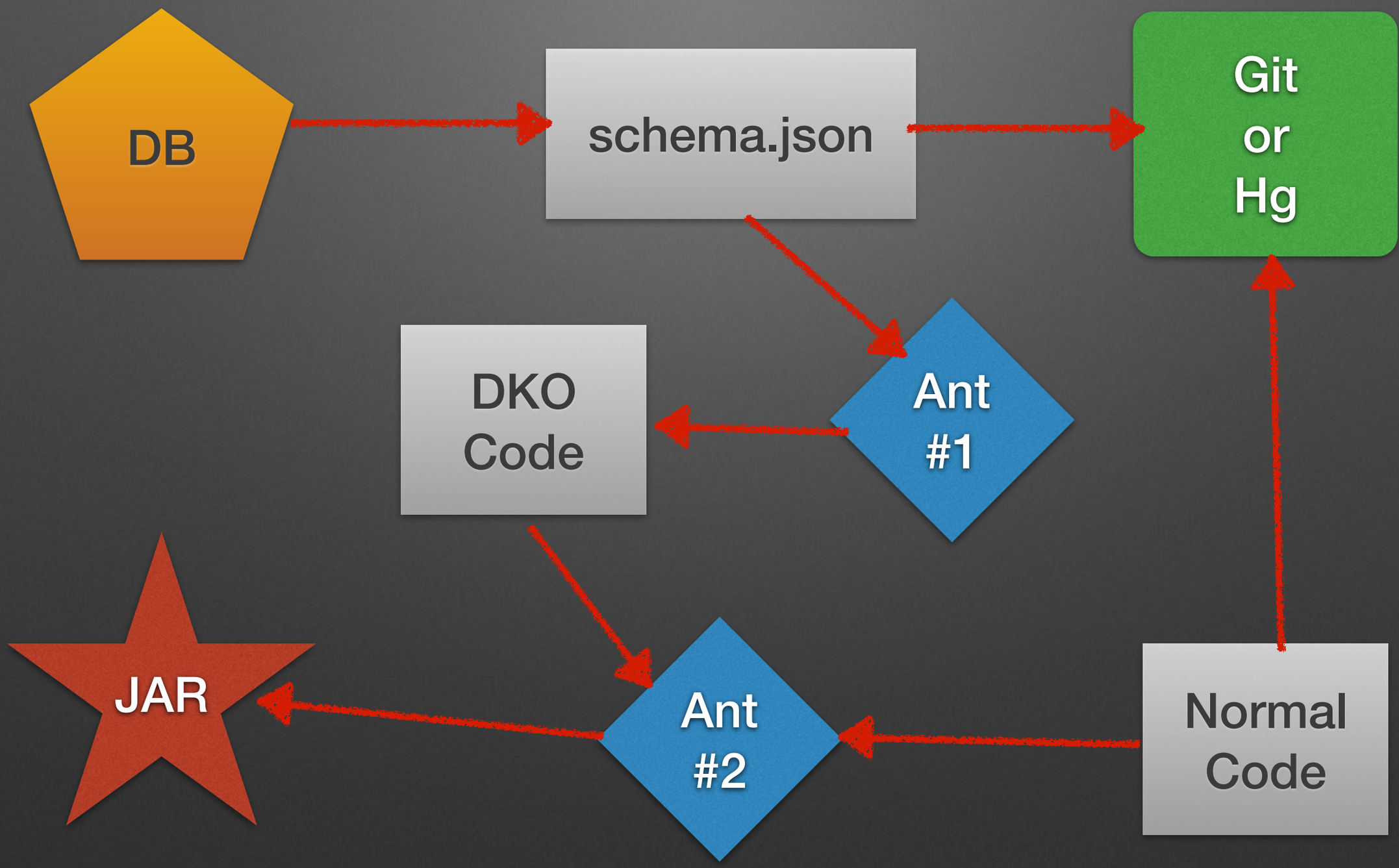
Can it be made non-evil?

Code Generation

Evil when:

- It's not automated.
- It requires external system states (like the database).
- People EDIT the fricking generated code!
  (What would your mother say?!?)
- People check it in to version control.
- It's unreadable / un-debuggable.

Code Generation



Performance

Considering JDBC as the gold standard:

testDKOCreate is 98.7% the speed of testJDBCCreate
testDKOSelect is 97.3% the speed of testJDBCSelect

Tests details:

20-run averages;
randomly ordered;
worst run dropped;
40,000 rows per run

All assuming equal queries…

Software Life Cycle

- If a table or column name changes, won't my next build fail?
  - Yes! This is the intent! That’s how you know what code needs to be checked!
  - Better a compiler error than a runtime error...
  - Use Eclipse's “rename” function to fix most renames. (fast, global and automatic)
  - Guaranteed coverage! (unlike grep)

Callbacks

Specify a callback package, and DKOs will automatically call InstrumentCB.preInsert(), postInsert(), preDelete(), postDelete(), etc. if they exist.

```
package com.company.dko.callbacks;
public class InstrumentCB {
    public static void preInsert(Instrument[] instrs) {
        // set auto-genned id values here
        // note: you get an array, not one at a time...
        // this allows you to bulk-get ids if you need to
        // poll an external service
    }
}
```

Current Status

Supports:

- Microsoft SQL Server
- MySQL / MariaDB
- PostgreSQL
- Oracle
- HSQL
- SQLite3
- Android (using SQLDroid)

Stats:

- ~13k Lines of code
- ~200 Unit tests
- > 3 years in production (11/2010)
- A major component of **REDACTED**'s data management.

https://github.com/keredson/DKO

http://nosco.googlecode.com/hg/doc/dkos-the-book/dkos-the-book.html