ERIC VERGNAUD

REIFYING ATTRIBUTES

INTRODUCTION

- Who am I?
  - 28 years in IT, developer, architect, manager
  - Startup founder, patent holder, OSS contributor
  - 9 years in very large and global organizations
- Why this talk?
  - Reifying attributes could solve issues identified in information systems
- What to expect?
  - Challenge status quo, improve practice

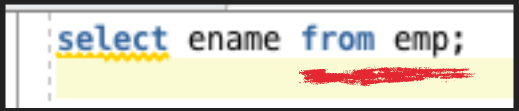
AGENDA

- What does reifying attributes mean?
- Reminder: unsharding the information system
- What problems do we have?
- Let’s revisit types, objects and inheritance
- Reifying attributes, through existing and experimental technologies
- Towards reified attributes using Java and SQL

PILLOWS

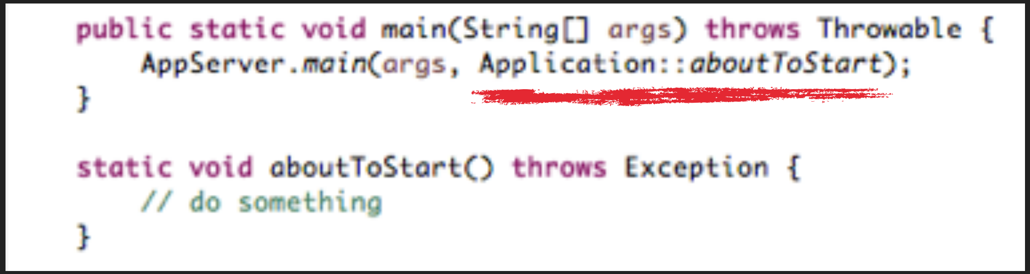
WHAT DOES REIFYING ATTRIBUTES MEAN?

- Attribute aka property, column, field
- Reification: « making something a first-class citizen » (wikipedia)
- In SQL, tables are reified, you can refer to them directly, but columns are not, they can only be referred to in the context of a table:



```
select ename from emp;
```

- In Java 8, functions are reified (to a certain extent), you can pass them as parameters:

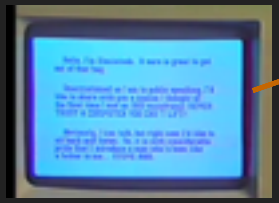


```
public static void main(String[] args) throws Throwable {
    AppServer.main(args, Application::aboutToStart);
}

static void aboutToStart() throws Exception {
    // do something
}
```

- Reifying attributes means making attributes first-class citizens, thus being able to refer to them globally, and use them as variables or parameters

UNSHARDING THE INFORMATION SYSTEM

- From mainframes to distributed computing
- Technology was an enabler, but the driver was cultural
  - Move away from centralized decisions
  - The 1984 ad, and the unveiling of the Macintosh

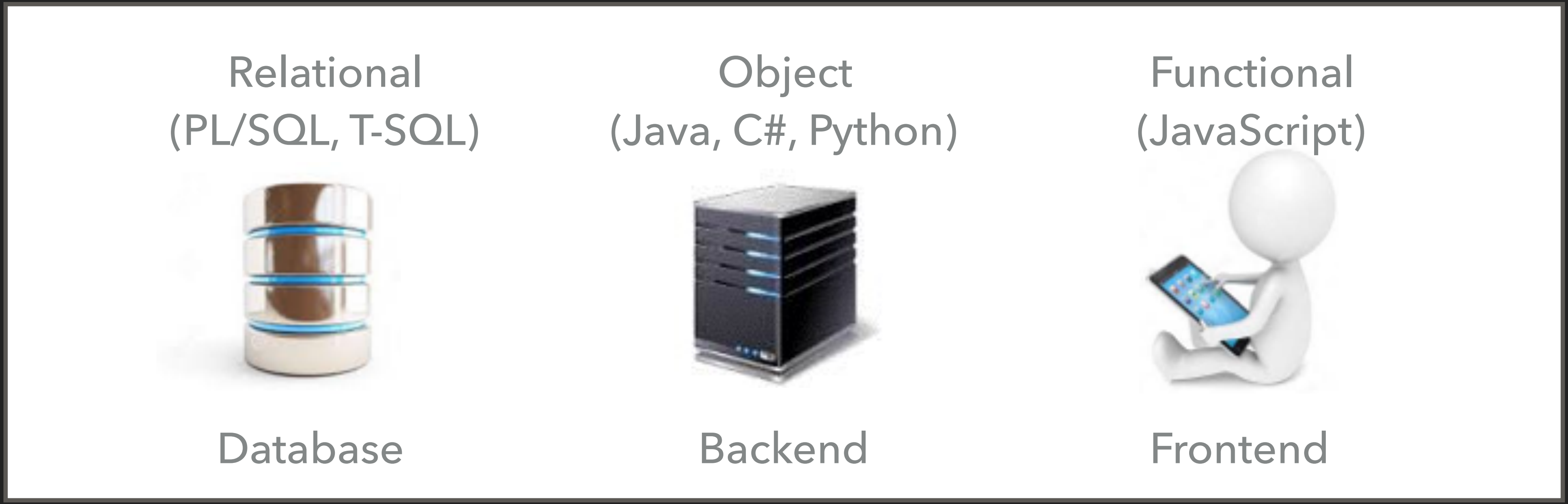


- There's no free lunch, distributed information systems have their trade-offs
  - Great to shard processes
  - Acceptable to shard data
  - Wrong to shard data models, because sharding models breaks them
- Big data potentially changes the game
- The « distributed mainframe »

WHAT PROBLEMS DO WE HAVE?

THE SHARDED DATA MODEL PROBLEM

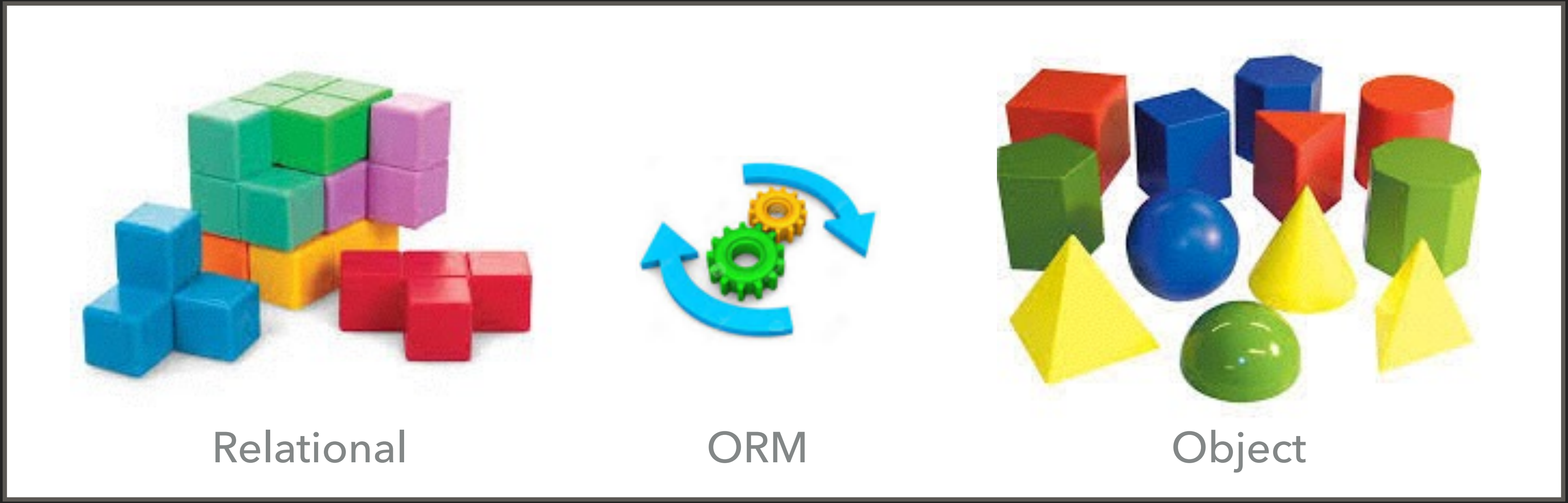
- Information system applications are built from various components: database, backend, front end
- Each of these components comes with its own data model paradigm, and language



- The variety of data modeling paradigms makes it impossible to guarantee the consistency of data models end to end

THE FAMOUS IMPEDANCE MISMATCH PROBLEM

- Popular databases are relational, objects are not, which is why we need and use ORMs (Hibernate, iBatis, JPA or custom built)



- Although less popular, a similar problem exists between backends and frontends.

THE FAMOUS IMPEDANCE MISMATCH PROBLEM

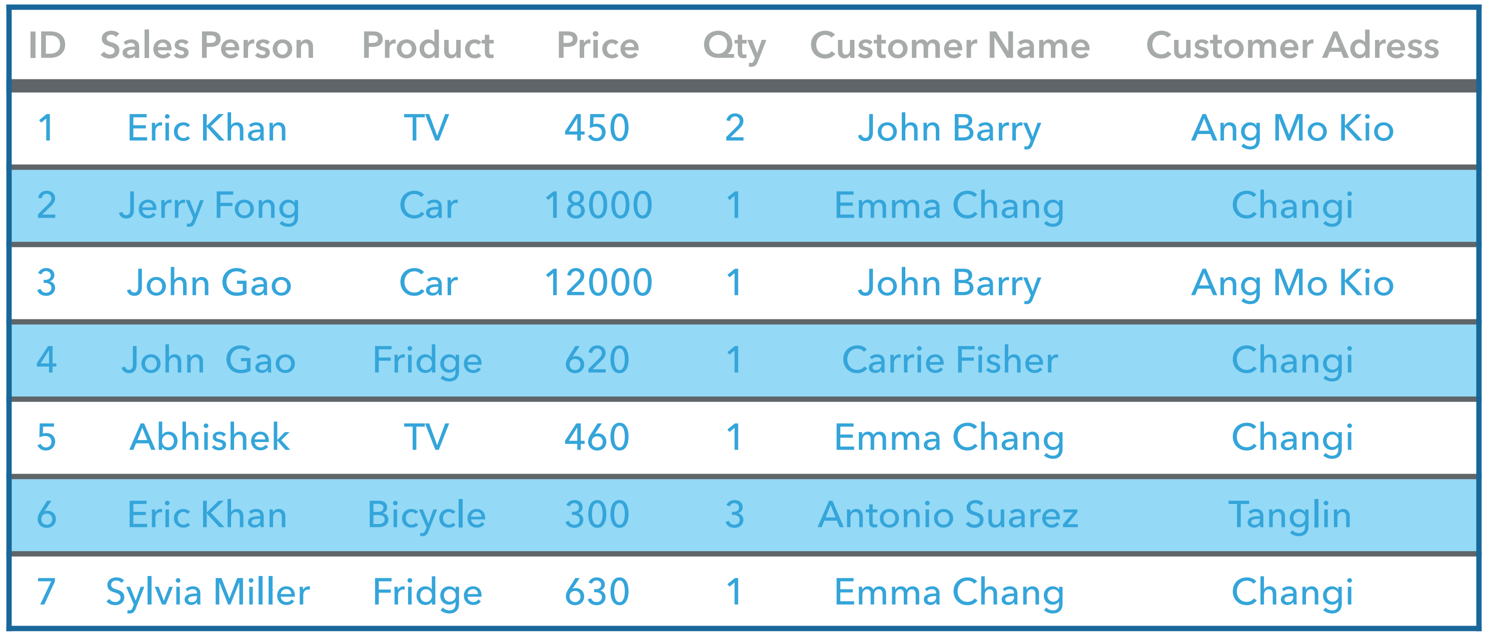
- Just because we seem to have a solution doesn't mean the problem has disappeared, it just makes it workable.
- ORMs don't solve the problem entirely
  - We still need to maintain multiple data models, 1 per tier
  - The database model is often shared across multiple applications, making it difficult to evolve without risk
  - The fronted programming language (JavaScript), the backend ones (Java, C#, Python) and the data manipulation language (SQL) do not naturally collaborate (although .Net LinQ is a major improvement)

IS THERE A DATA MODELING PARADIGM WE COULD USE FOR ALL TIERS OF AN INFORMATION SYSTEM APPLICATION?

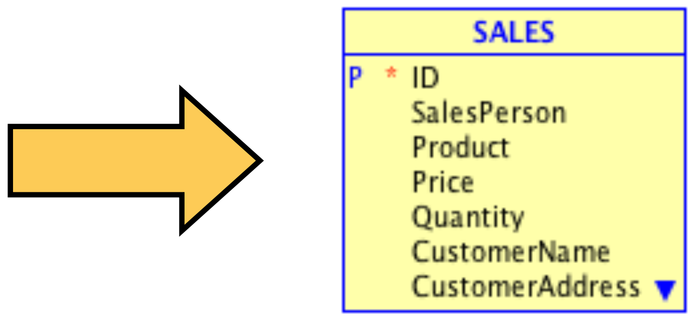
COULD REIFIED ATTRIBUTES BE THAT DATA MODELING PARADIGM?

THE NON RELATIONAL META RELATIONAL MODEL PROBLEM

- A quick refresh of the 3 Boyce-Codd Normal Forms, through a bad example



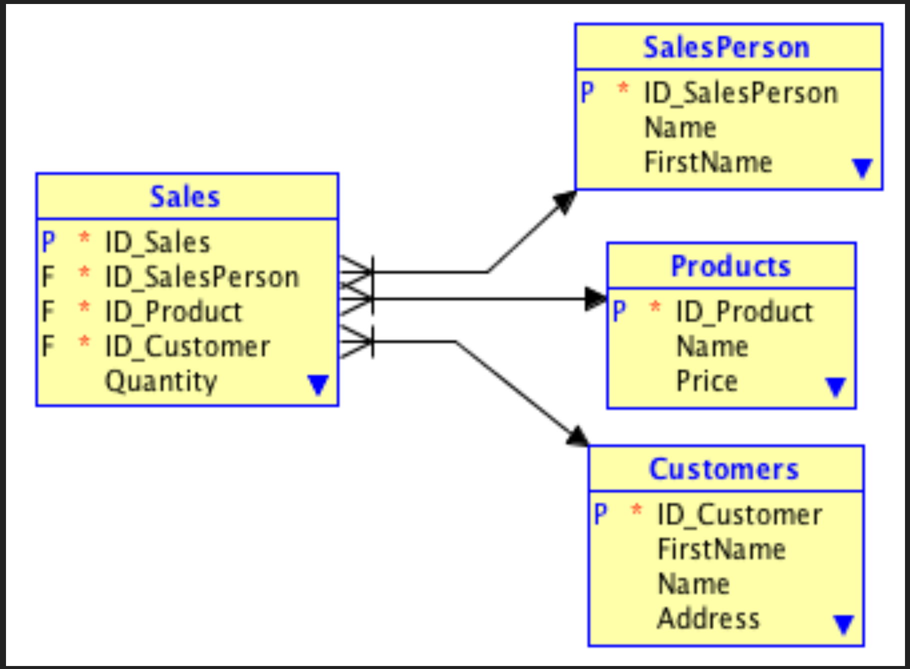
ID	Sales Person	Product	Price	Qty	Customer Name	Customer Adress
1	Eric Khan	TV	450	2	John Barry	Ang Mo Kio
2	Jerry Fong	Car	18000	1	Emma Chang	Changi
3	John Gao	Car	12000	1	John Barry	Ang Mo Kio
4	John Gao	Fridge	620	1	Carrie Fisher	Changi
5	Abhishek	TV	460	1	Emma Chang	Changi
6	Eric Khan	Bicycle	300	3	Antonio Suarez	Tanglin
7	Sylvia Miller	Fridge	630	1	Emma Chang	Changi



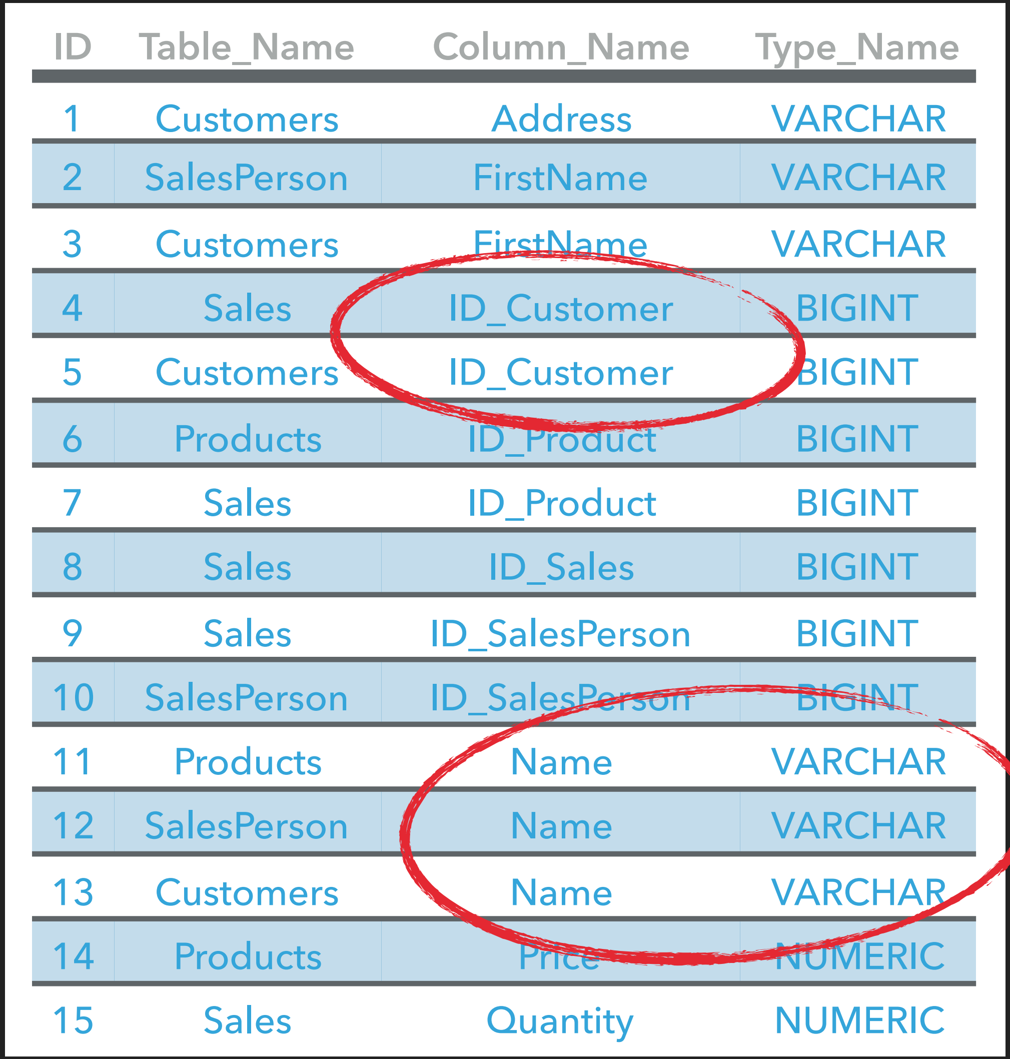
- The above actually breaks the 3 normal forms:
  - Sales Person and Customer Name each contain 2 atomic values, breaks 1NF
  - Customer Address depends on Customer Name, which is not the primary key, breaks 2NF
  - Sales Person, Customer Name and Customer Address have duplicates, breaks 3NF

THE NON RELATIONAL META RELATIONAL MODEL PROBLEM

- A 3NF compliant model could look like the below



- But what about its meta-model, i.e. the model used to describe the relational model itself? Is it 3NF compliant ?



ID	Table_Name	Column_Name	Type_Name
1	Customers	Address	VARCHAR
2	SalesPerson	FirstName	VARCHAR
3	Customers	FirstName	VARCHAR
4	Sales	ID_Customer	BIGINT
5	Customers	ID_Customer	BIGINT
6	Products	ID_Product	BIGINT
7	Sales	ID_Product	BIGINT
8	Sales	ID_Sales	BIGINT
9	Sales	ID_SalesPerson	BIGINT
10	SalesPerson	ID_SalesPerson	BIGINT
11	Products	Name	VARCHAR
12	SalesPerson	Name	VARCHAR
13	Customers	Name	VARCHAR
14	Products	Price	NUMERIC
15	Sales	Quantity	NUMERIC

RELATIONAL IS ABOUT COST OF HARDWARE, NOT CORRECTNESS

- A data model must comply with 3NF to be *said relational*.
- But a data model does not have to be relational to be correct (think of the object model)
- Actually the purpose of the relational model was to:
  - limit storage size by avoiding data duplication
  - limit processing cost of multiple updates
  - reduce risk of inconsistencies
- The purpose was to limit hardware cost, at a time where manpower was cheaper than hardware

COST OF HARDWARE IS NO LONGER A CONCERN

- Original purpose of the relational model:
  - limit storage size by avoiding data duplication
  - limit processing cost of multiple updates
  - reduce risk of inconsistencies
- The 2 former were perfectly valid objectives 30 years ago when engineering was cheaper than hardware, but is that still the case today with hardware way cheaper than engineering?
- Systems we actually build often seem to not care much about storage cost: we duplicate data for audit purpose, and we denormalize it for performance

IS THERE A DATA MODELING PARADIGM BETTER SUITED TO THE NEW BALANCE OF IT COSTS?

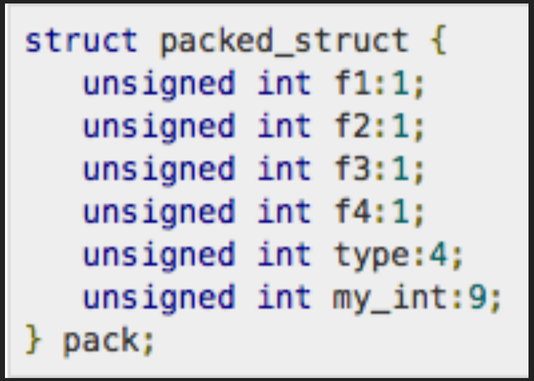
DISCUSSION

REVISITING TYPES

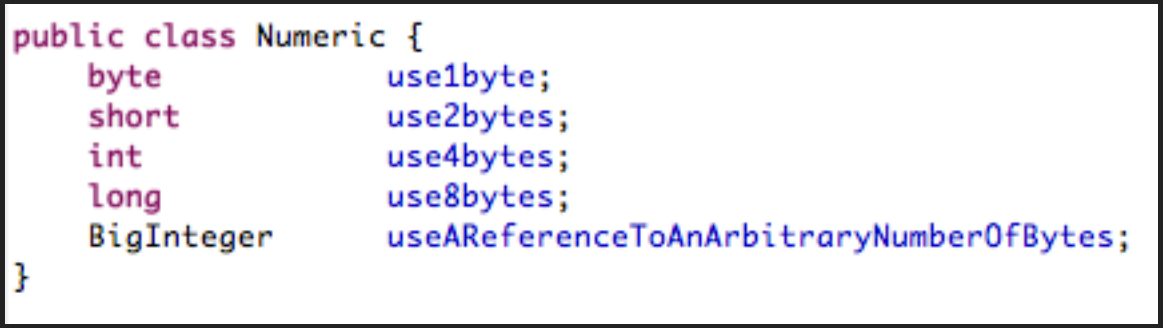
WHAT IS THE TYPE OF THIS THING (CALLED A CHAIR)?

HARDWARE ORIENTED TYPES

- What do we mean, in C, by « `unsigned short a[3] = {2,3,7};` » ?
  - allocate exactly 48 bits of RAM (from the heap or the stack depending on context)
  - treat them as 3 chunks of 16 bits each
  - treat each chunk as an integer >=0 & <= 65535 (unsigned short)
  - assign unsigned short values 2, 3 and 7 to the 3 chunks
- The Java and C# type systems mimic the C type system i.e. they require focus on hardware resources allocation



```
struct packed_struct {
   unsigned int f1:1;
   unsigned int f2:1;
   unsigned int f3:1;
   unsigned int f4:1;
   unsigned int type:4;
   unsigned int my_int:9;
} pack;
```

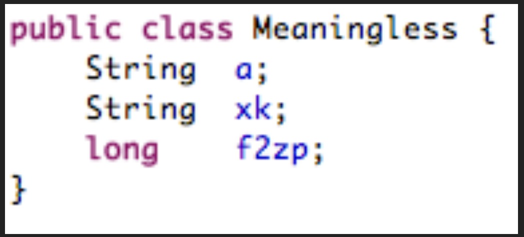


```
public class Numeric {
    byte            use1byte;
    short           use2bytes;
    int             use4bytes;
    long            use8bytes;
    BigInteger      useAReferenceToAnArbitraryNumberOfBytes;
}
```

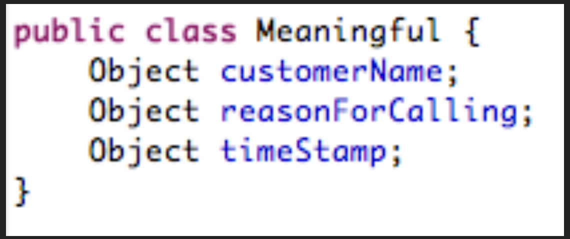
- This enables *interesting* conversations:
  - how many units were sold?
    - do you want the answer as a byte, a short, an int, a long or a reference to an arbitrary number of bytes?

USAGE ORIENTED TYPES

- What do we mean, in Python, by « `a = [3, 11.2, -12]` » ?
  - allocate a list of 3 elements with values 3, 11.2 and -12
  - we have no idea on what hardware resources have been allocated
- Dynamic typing is not only about deciding type at runtime, it's also about not caring so much about hardware resources allocation (leave that to the runtime), and rather focus on meaning
  - Python will adjust the allocation as required, when the value changes
  - Python pre-allocates integers from -5 to +256 for performance reason
- When dealing with information, meaning is almost always more important than hardware resources allocation:



```java
public class Meaningless {
    String  a;
    String  xk;
    long    f2zp;
}
```



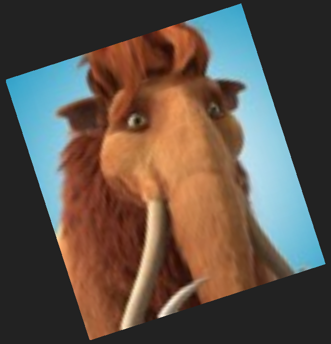
```java
public class Meaningful {
    Object customerName;
    Object reasonForCalling;
    Object timeStamp;
}
```

WHAT IS THE TYPE OF THIS THING (CALLED A CHAIR)?

- In an information system, a type should describe usage i.e. how something is used, not structure i.e. how something is built

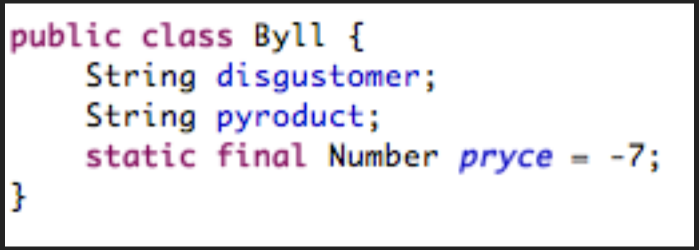
- Can reifying attributes help us focus on meaning rather than structure?

IMAGINARY TYPES vs TYPES ABOUT REALITY

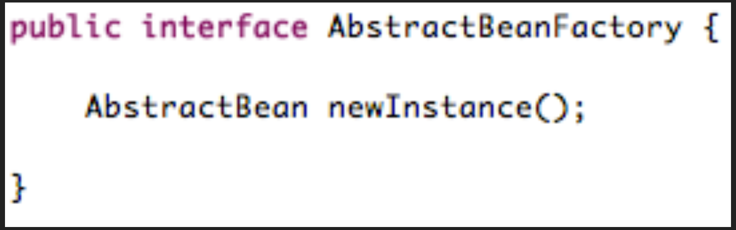


Ice age Ellie thinks she's a possum (she's actually wrong)

- The concept of « type » is itself imaginary: a stone doesn't know it's a stone, a bird probably doesn't know it's a bird
- When we define the « bird » type, we're actually defining a type about something real. The type « bird » describes a number of real characteristics (have feathers and wings, lay eggs…) which put together uniquely define a specie. Those characteristics are *observed*, not *designed* by humans. They are not negotiable.
- This is different from imaginary types, such as the below :



```
public class Byll {
    String disgustomer;
    String pyroduct;
    static final Number pryce = -7;
}
```



```
public interface AbstractBeanFactory {

    AbstractBean newInstance();

}
```

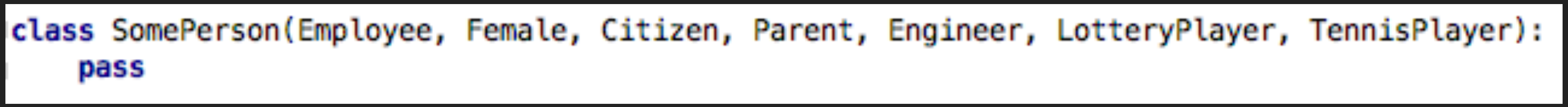
INHERITANCE AND THE DEADLY DIAMOND

- It is close to impossible to model the real world using Java/C# single inheritance:

- Types are not reality, they are a restriction of reality

INHERITANCE AND THE FAMOUS DEADLY DIAMOND

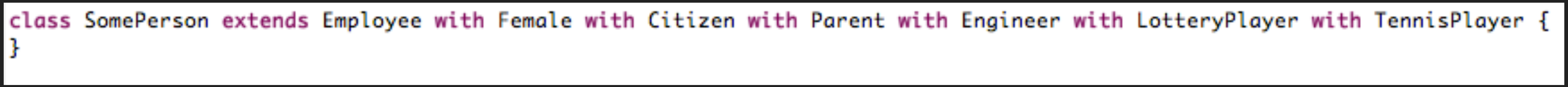
- Python supports multiple inheritance:



```
class SomePerson(Employee, Female, Citizen, Parent, Engineer, LotteryPlayer, TennisPlayer):
    pass
```

there is no guarantee that the data in Citizen and Engineer have the same meaning

- Scala supports multiple inheritance:



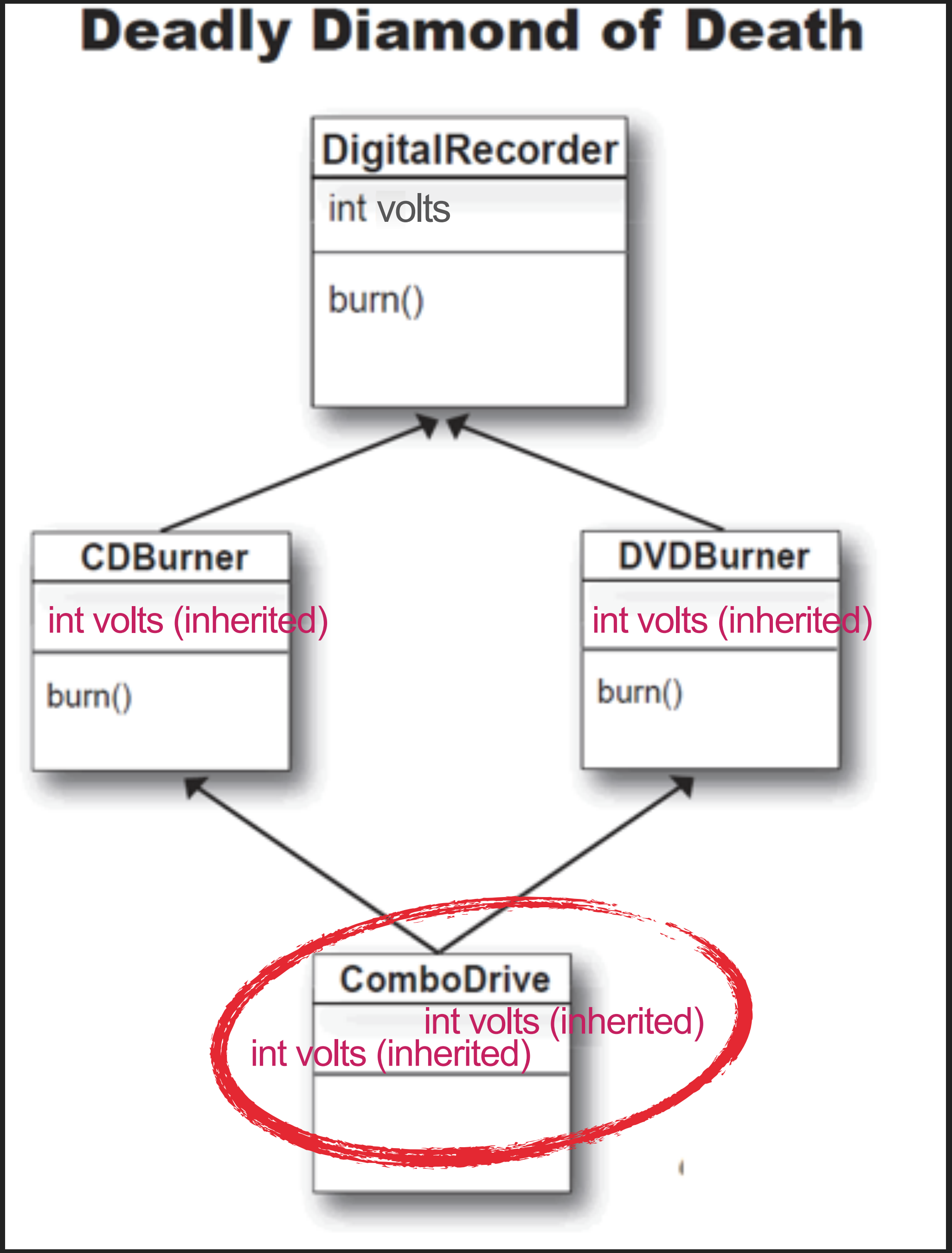
```
class SomePerson extends Employee with Female with Citizen with Parent with Engineer with LotteryPlayer with TennisPlayer {
}
```

only behavior is supported by multiple inheritance
data needs to be implemented by the class or its parent

- So why does Java not support multiple inheritance?

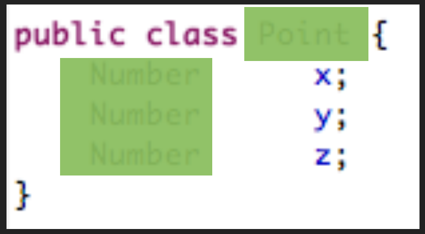
INHERITANCE AND THE FAMOUS DEADLY DIAMOND

- Both CDBurner and DVDBurner inherit member « `int volts` »
- ComboDrive inherits it twice!!!
- The virtual table helps decide which « burn() » method to call for ComboDrive, but no mechanism can decide which « `int volts` » member to use.
- Python does not suffer from the problem, because it doesn't care about storage at « compile » time.

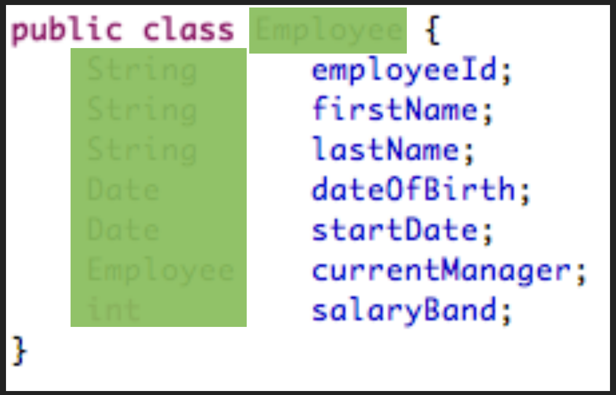


TYPES ARE INFERRED BY ATTRIBUTES, NOT THE OPPOSITE

- Challenge: what would be the types of the below classes?



```
public class Point {
    Number      x;
    Number      y;
    Number      z;
}
```



```
public class Employee {
    String          employeeId;
    String          firstName;
    String          lastName;
    Date            dateOfBirth;
    Date            startDate;
    Employee        currentManager;
    int             salaryBand;
}
```

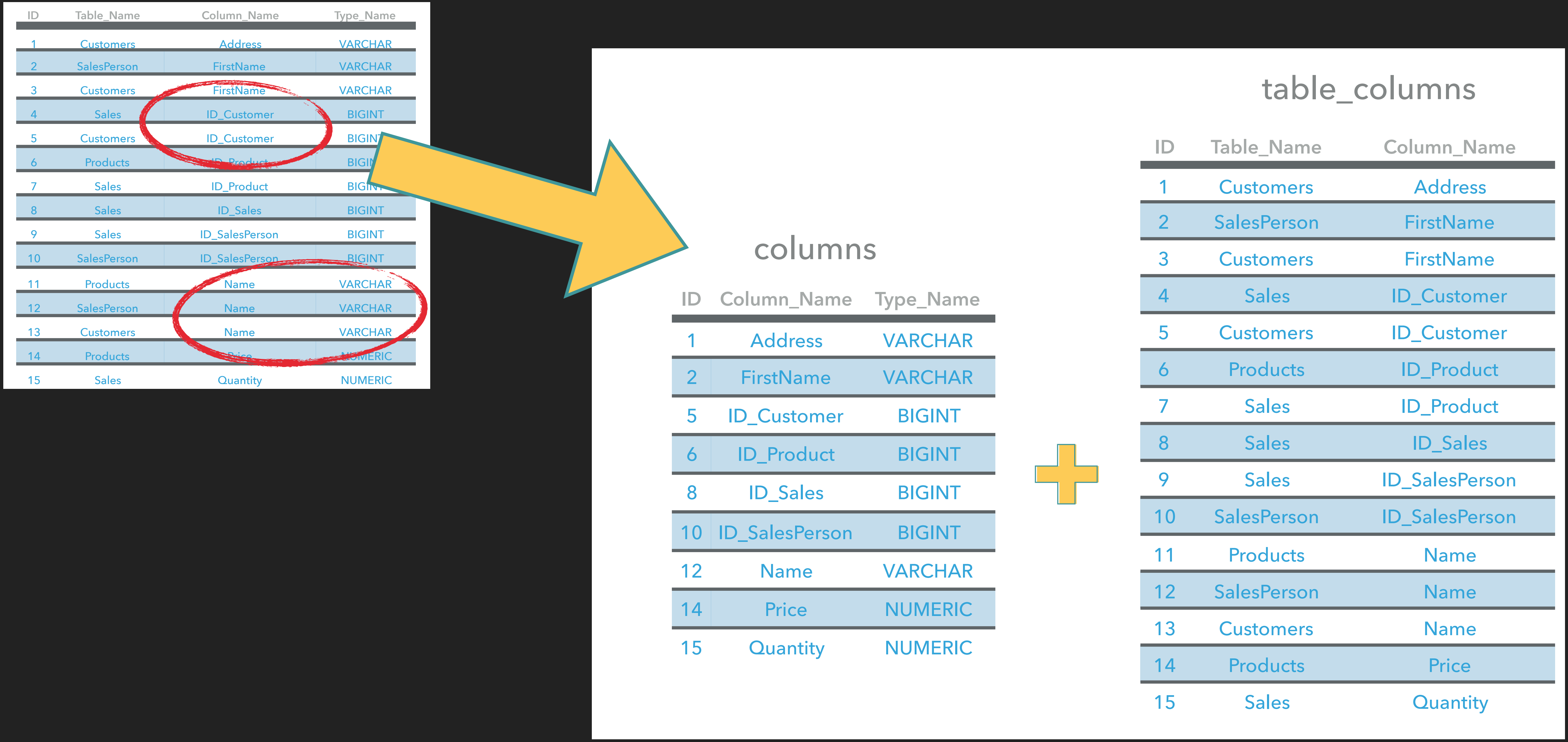
- Challenge: what would be the types of the class members?
- Maybe attributes are self-sufficient, and deserve to be reified?

DISCUSSION

REIFYING ATTRIBUTES

WHAT WOULD REIFYING ATTRIBUTES MEAN FOR RELATIONAL DATABASES?

- Columns live outside tables, and they have types



ID	Table_Name	Column_Name	Type_Name
1	Customers	Address	VARCHAR
2	SalesPerson	FirstName	VARCHAR
3	Customers	FirstName	VARCHAR
4	Sales	ID_Customer	BIGINT
5	Customers	ID_Customer	BIGIN
6	Products	ID_Product	BIGI
7	Sales	ID_Product	BIGINT
8	Sales	ID_Sales	BIGINT
9	Sales	ID_SalesPerson	BIGINT
10	SalesPerson	ID_SalesPerson	BIGINT
11	Products	Name	VARCHAR
12	SalesPerson	Name	VARCHAR
13	Customers	Name	VARCHAR
14	Products	Price	NUMERIC
15	Sales	Quantity	NUMERIC

columns

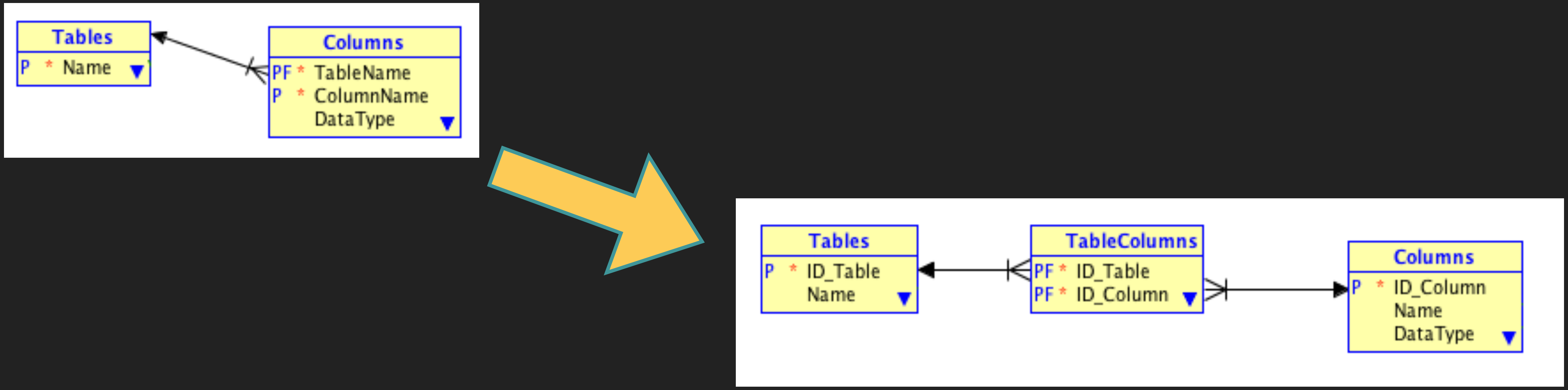
ID	Column_Name	Type_Name
1	Address	VARCHAR
2	FirstName	VARCHAR
5	ID_Customer	BIGINT
6	ID_Product	BIGINT
8	ID_Sales	BIGINT
10	ID_SalesPerson	BIGINT
12	Name	VARCHAR
14	Price	NUMERIC
15	Quantity	NUMERIC

table_columns

ID	Table_Name	Column_Name
1	Customers	Address
2	SalesPerson	FirstName
3	Customers	FirstName
4	Sales	ID_Customer
5	Customers	ID_Customer
6	Products	ID_Product
7	Sales	ID_Product
8	Sales	ID_Sales
9	Sales	ID_SalesPerson
10	SalesPerson	ID_SalesPerson
11	Products	Name
12	SalesPerson	Name
13	Customers	Name
14	Products	Price
15	Sales	Quantity

WHAT WOULD REIFYING ATTRIBUTES MEAN FOR RELATIONAL DATABASES?

- Minimal changes in the relational model:
  - no change in tables
  - columns are picked from global column definitions
  - column data types are specified once
- The relational meta-model would change as follows:



HOW NEW IS THIS?

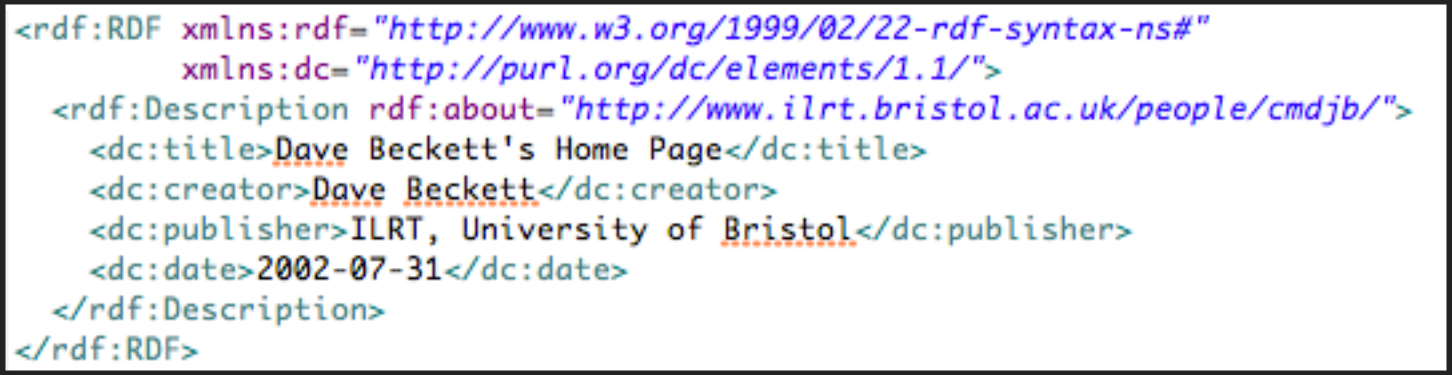
- Such « reified » attributes already exist in relational databases:
  - Oracle's ROWID and ORA_ROWSCN
  - SQL Server's $ROWGUIDCOL
- They are centric to the semantic web and the RDF model
- They are important components of graph databases such as Neo4J
- They are at the core of Datomic, a new kind of database

THE SEMANTIC WEB AND THE RDF MODEL

- After Tim Berners-Lee (inventor of Internet and head of w3c) wrote an article describing the need for a semantic web, the w3c proposed the Resource Description Framework (RDF) model, which is now widely used.
- The general principle is that any meaningful data about a resource, identified by its location, can be described using « triples » in the form:

```
locator-predicate-(value or resource)
```

- Example:

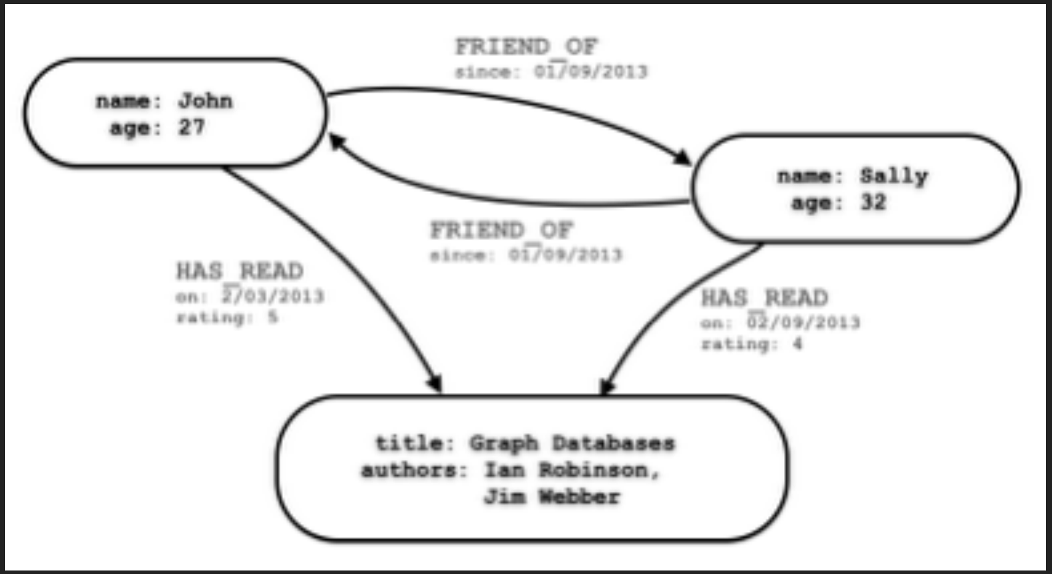


```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description rdf:about="http://www.ilrt.bristol.ac.uk/people/cmdjb/">
    <dc:title>Dave Beckett's Home Page</dc:title>
    <dc:creator>Dave Beckett</dc:creator>
    <dc:publisher>ILRT, University of Bristol</dc:publisher>
    <dc:date>2002-07-31</dc:date>
  </rdf:Description>
</rdf:RDF>
```

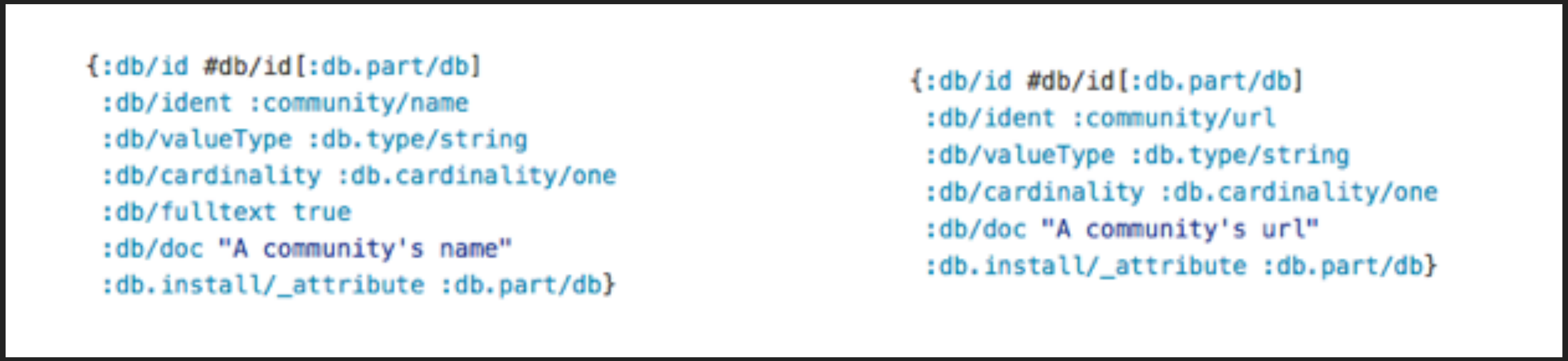
- The key point here for us is that predicates i.e. attributes are defined outside any category/class/table context. « `dc:creator` » has the same meaning whatever is being described, and is defined once and for all in the « Dublin Core » specification. They are reified.

GRAPH DATABASES AND DATOMIC

- An example from Neo4J. Properties (name, age) can be attached to any node, regardless of their « type »:



- Datomic follows the same paradigm, letting you defined typed attributes outside any context (the namespace is optional):



```
{:db/id #db/id[:db.part/db]
 :db/ident :community/name
 :db/valueType :db.type/string
 :db/cardinality :db.cardinality/one
 :db/fulltext true
 :db/doc "A community's name"
 :db.install/_attribute :db.part/db}
```

```
{:db/id #db/id[:db.part/db]
 :db/ident :community/url
 :db/valueType :db.type/string
 :db/cardinality :db.cardinality/one
 :db/doc "A community's url"
 :db.install/_attribute :db.part/db}
```

THE END OF TABLES?

- Since reified attributes live outside tables, any « table » is simply a collection of attributes. If any entity can be assigned any attribute, then a table is the collection of all attributes. As a consequence, all tables are the same i.e. there is only one table. The entity type is then just another attribute.

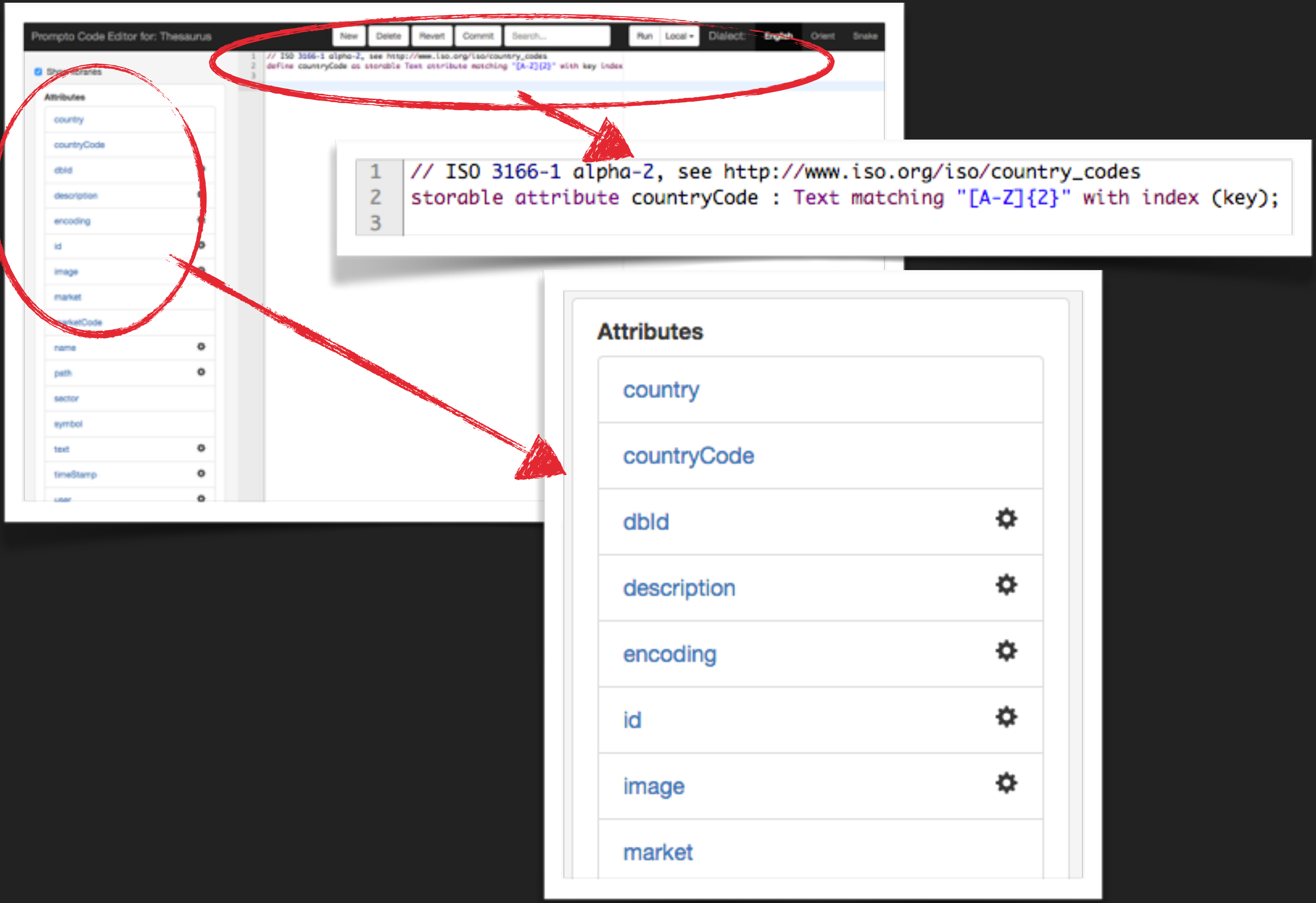
- Relational databases are clearly not good candidates. A single table with hundreds of attributes largely populated with nulls is likely to be both costly in terms of disk space and very slow.

- Conversely, NoSQL databases, where the structure of each row is loose, are pretty good candidates: HBase, Datomic, Neo4J, MongoDB, SOLR and any schema-less or lightweight-schema database engine.

WHAT ABOUT PROGRAMMING LANGUAGES?

- Reifying attributes can also be applied to programming languages.
- It's a way to unify data modeling across solution tiers. If the same attributes can be used in every tier, then there is indeed only one data model.
- Prompto is an experimental language which explores this strategy (also explores unrelated matters such as deployment and collaboration).
- The language is not available (yet)

THE PROMPTO LANGUAGE: DEFINING ATTRIBUTES



```
// ISO 3166-1 alpha-2, see http://www.iso.org/iso/country_codes
define countryCode as storable Text attribute matching "[A-Z]{2}" with key index
```

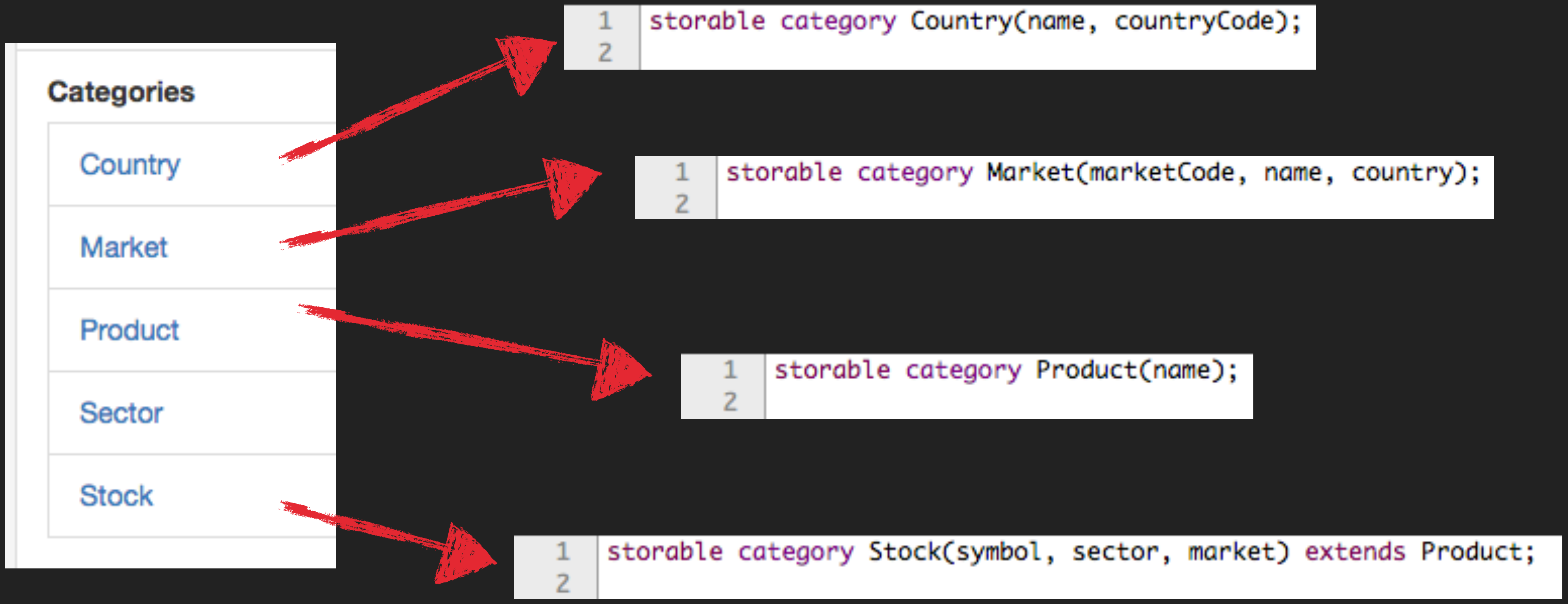
```
// ISO 3166-1 alpha-2, see http://www.iso.org/iso/country_codes
storable attribute countryCode : Text matching "[A-Z]{2}" with index (key);
```

Attributes

- country
- countryCode
- dbId
- description
- encoding
- id
- image
- market
- marketCode
- name
- path
- sector
- symbol
- text
- timeStamp
- user

THE PROMPTO LANGUAGE: DEFINING CATEGORIES

- Categories are defined from a subset of attributes picked from the list of all attributes.



```
storable category Country(name, countryCode);
```

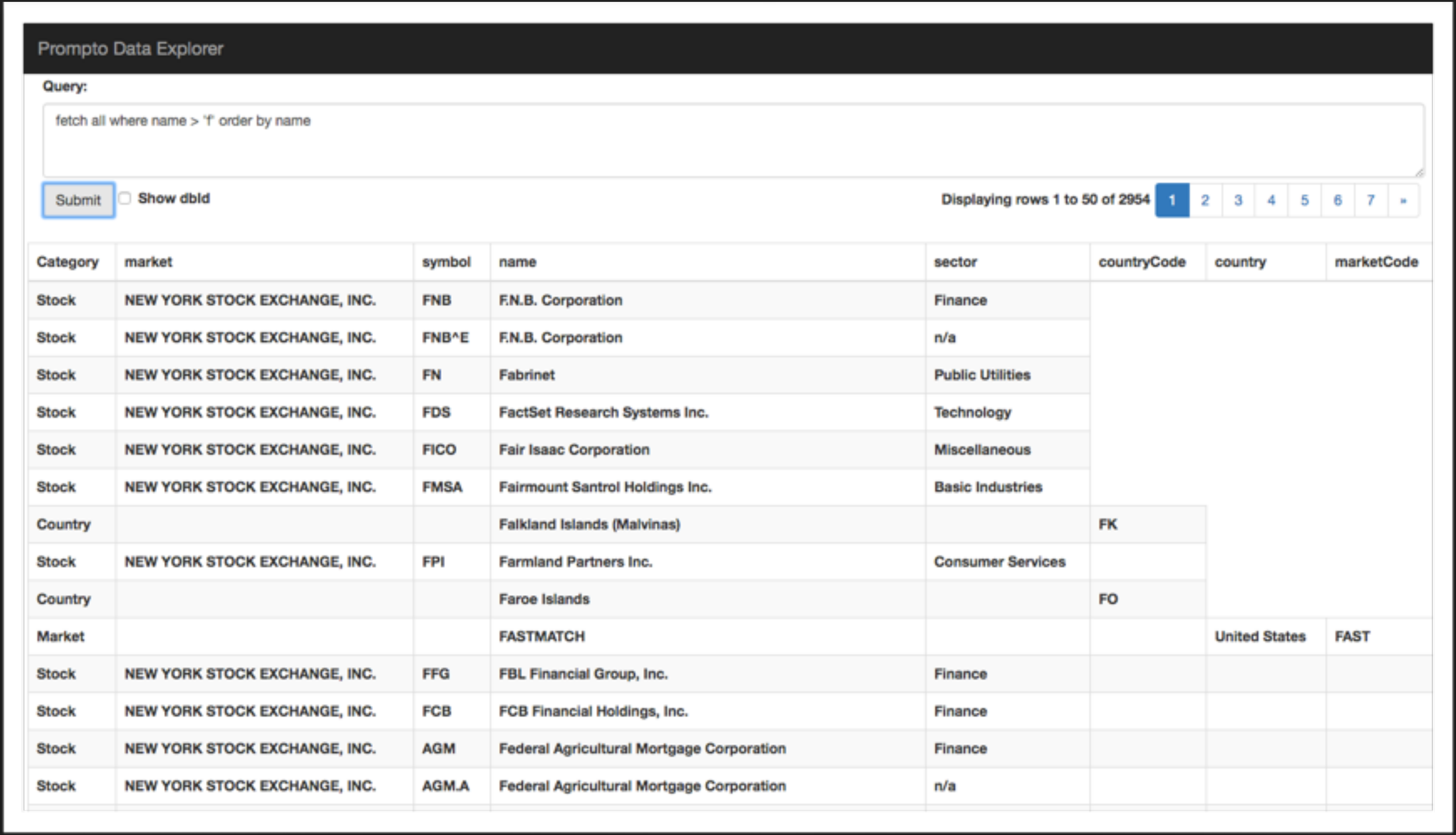
```
storable category Market(marketCode, name, country);
```

```
storable category Product(name);
```

```
storable category Stock(symbol, sector, market) extends Product;
```

ENABLES NEW QUERYING CAPABILITIES

Search can be achieved on all the data using one query



Prompto Data Explorer

Query:

```
fetch all where name > 'f' order by name
```

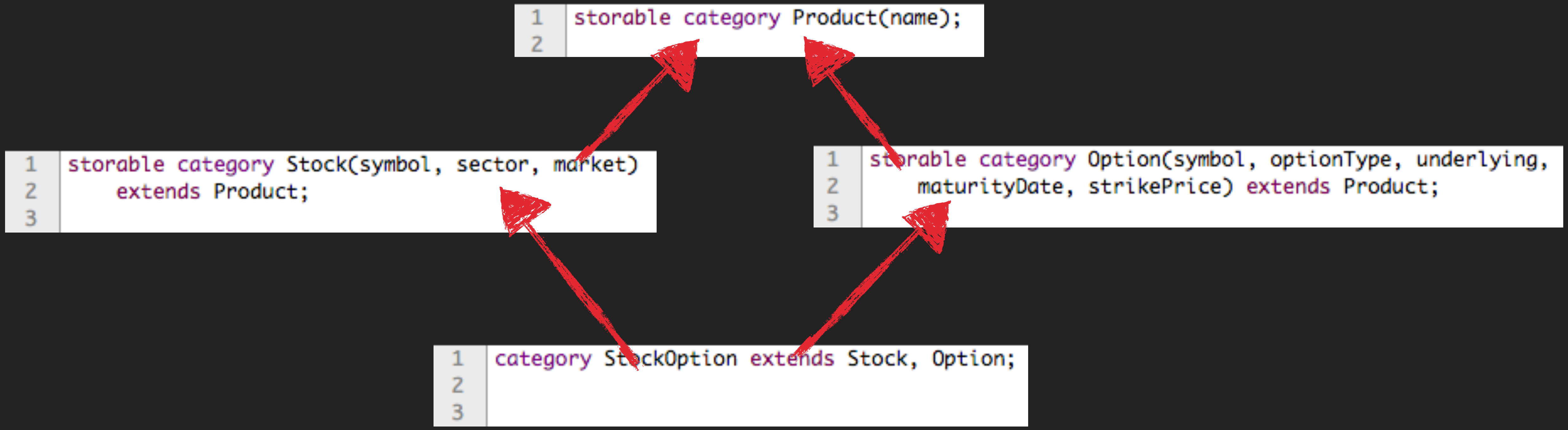
Submit ☐ Show dbId

Displaying rows 1 to 50 of 2954 1 2 3 4 5 6 7 »

Category	market	symbol	name	sector	countryCode	country	marketCode
Stock	NEW YORK STOCK EXCHANGE, INC.	FNB	F.N.B. Corporation	Finance			
Stock	NEW YORK STOCK EXCHANGE, INC.	FNB^E	F.N.B. Corporation	n/a			
Stock	NEW YORK STOCK EXCHANGE, INC.	FN	Fabrinet	Public Utilities			
Stock	NEW YORK STOCK EXCHANGE, INC.	FDS	FactSet Research Systems Inc.	Technology			
Stock	NEW YORK STOCK EXCHANGE, INC.	FICO	Fair Isaac Corporation	Miscellaneous			
Stock	NEW YORK STOCK EXCHANGE, INC.	FMSA	Fairmount Santrol Holdings Inc.	Basic Industries			
Country			Falkland Islands (Malvinas)		FK		
Stock	NEW YORK STOCK EXCHANGE, INC.	FPI	Farmland Partners Inc.	Consumer Services			
Country			Faroe Islands		FO		
Market			FASTMATCH			United States	FAST
Stock	NEW YORK STOCK EXCHANGE, INC.	FFG	FBL Financial Group, Inc.	Finance			
Stock	NEW YORK STOCK EXCHANGE, INC.	FCB	FCB Financial Holdings, Inc.	Finance			
Stock	NEW YORK STOCK EXCHANGE, INC.	AGM	Federal Agricultural Mortgage Corporation	Finance			
Stock	NEW YORK STOCK EXCHANGE, INC.	AGM.A	Federal Agricultural Mortgage Corporation	n/a			

ELIMINATES THE DEADLY DIAMOND PROBLEM

- Since attributes are reified, the « name » attribute has the same role and meaning in all categories



```
storable category Product(name);
```

```
storable category Stock(symbol, sector, market)
    extends Product;
```

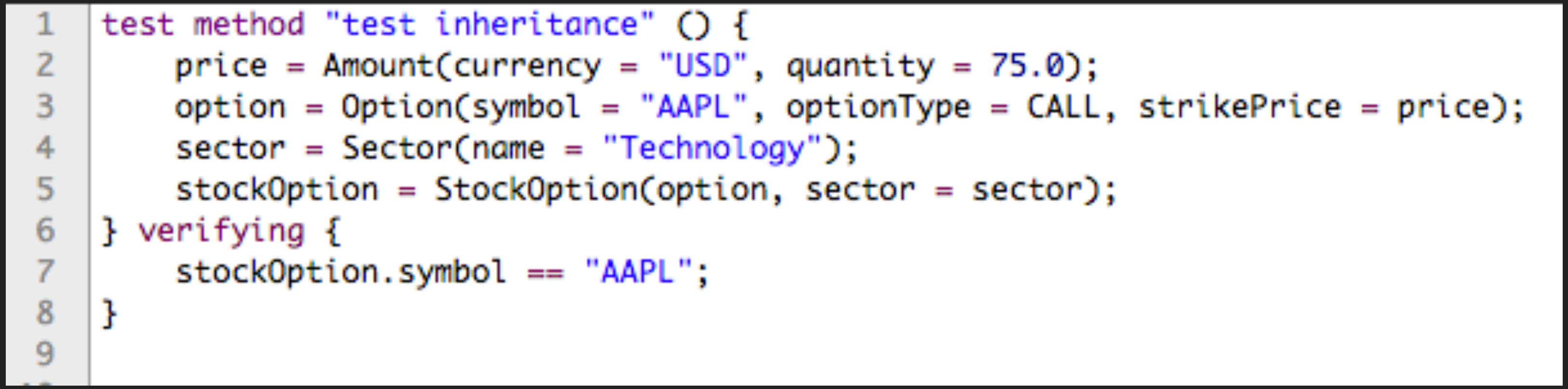
```
storable category Option(symbol, optionType, underlying,
    maturityDate, strikePrice) extends Product;
```

```
category StockOption extends Stock, Option;
```

- Similarly, the « symbol » attribute used for Stock is the same as the « symbol » attribute used for Option, so it's the same in the StockOption category
- For methods, the language uses a « first found wins » algorithm, similar to Python's MRO, and methods can be overridden

ENABLES AUTOMATIC COPY CONSTRUCTORS

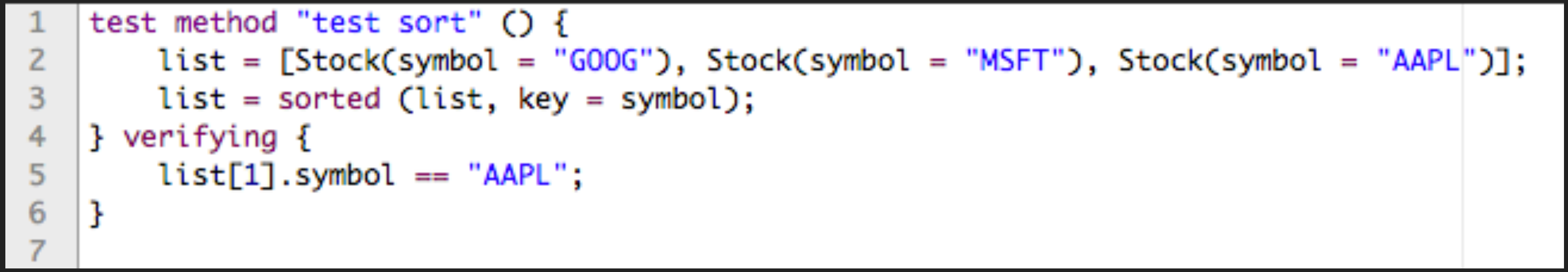
- Attributes can be copied automatically from a source instance to a new one



```
test method "test inheritance" () {
    price = Amount(currency = "USD", quantity = 75.0);
    option = Option(symbol = "AAPL", optionType = CALL, strikePrice = price);
    sector = Sector(name = "Technology");
    stockOption = StockOption(option, sector = sector);
} verifying {
    stockOption.symbol == "AAPL";
}
```

ENABLES SORT BY ATTRIBUTE

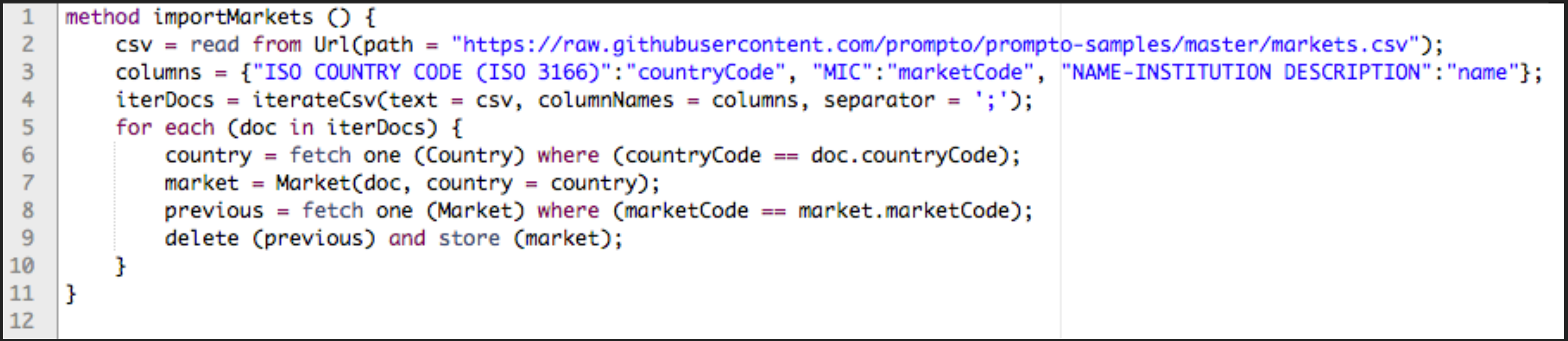
- Attributes are first class citizens, so they can be used as parameters



```
test method "test sort" () {
    list = [Stock(symbol = "GOOG"), Stock(symbol = "MSFT"), Stock(symbol = "AAPL")];
    list = sorted (list, key = symbol);
} verifying {
    list[1].symbol == "AAPL";
}
```

ENABLES STORE AND FETCH IN PLAIN CODE

- Attributes are used to define both the database model and the object model, the need to translate is eliminated



```
method importMarkets () {
    csv = read from Url(path = "https://raw.githubusercontent.com/prompto/prompto-samples/master/markets.csv");
    columns = {"ISO COUNTRY CODE (ISO 3166)":"countryCode", "MIC":"marketCode", "NAME-INSTITUTION DESCRIPTION":"name"};
    iterDocs = iterateCsv(text = csv, columnNames = columns, separator = ';');
    for each (doc in iterDocs) {
        country = fetch one (Country) where (countryCode == doc.countryCode);
        market = Market(doc, country = country);
        previous = fetch one (Market) where (marketCode == market.marketCode);
        delete (previous) and store (market);
    }
}

```

ENABLES ONE DATA MODEL END TO END

- The data model is defined once using attributes
- Attributes are used to populate the store schema if required
- Prompto runs in Java, C#, Python and JavaScript

DISCUSSION

REIFYING ATTRIBUTES IN SQL AND JAVA

WE CAN'T CHANGE RELATIONAL DATABASE ENGINES INTERNALS

THE BII…IIIG SQL TABLE

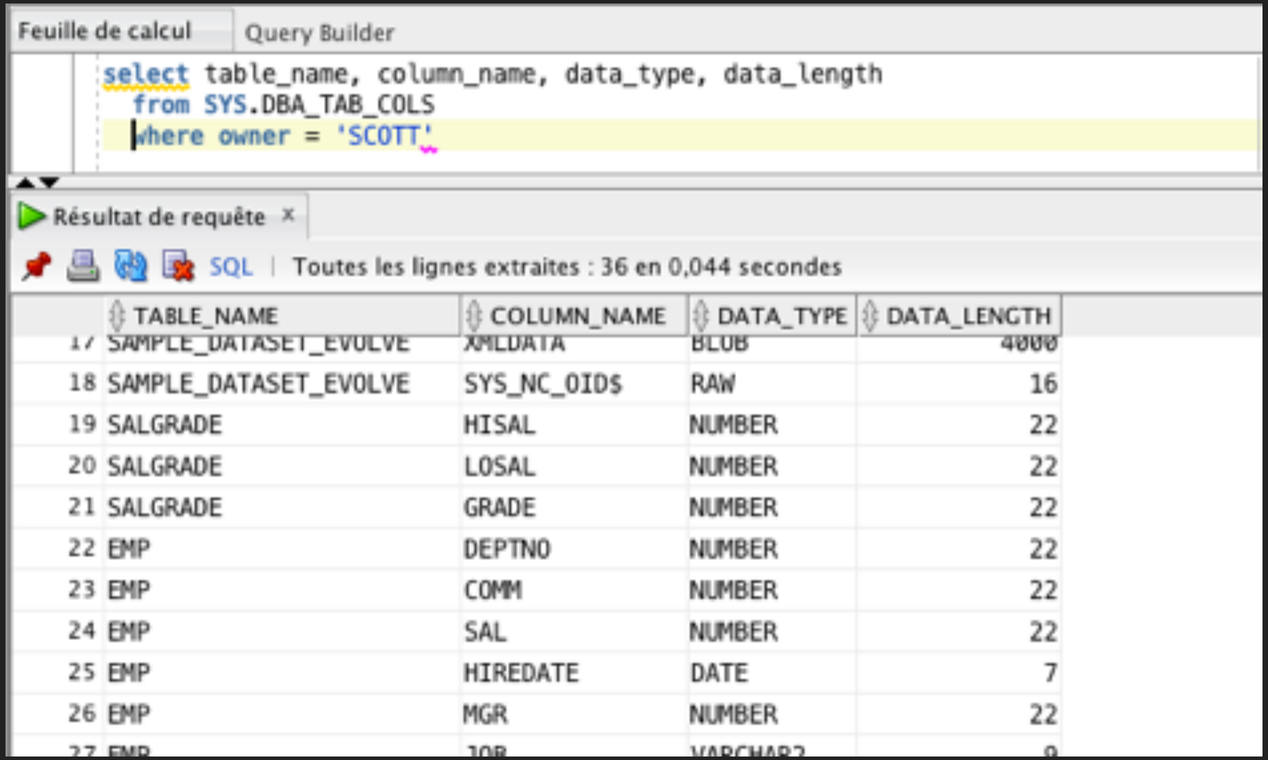
- A strategy that comes to mind is to have just one table, with hundreds or even thousands of columns. Probably not a good idea:
    - SQL databases store NULL values, so the table would be mostly empty
    - No support for multi-valued columns, can't store categories with each row
    - Table might end up being locked for updates most of the time
    - Not designed for purpose, can't expect support

THE COLUMNS VERIFIER STRATEGY

- Provided that columns always have the same type, « duplicating » them in tables can be considered as an implementation detail
- We can't easily detect column naming inconsistencies (id_order vs order_id, currency vs ccy)
- But we can detect type inconsistencies

DETECTING SQL COLUMN TYPES INCONSISTENCIES

- Model metadata is generally accessible (but there is no standard, each database requires specific code).
- Extracting relevant metadata from Oracle is very simple



Feuille de calcul | Query Builder

```
select table_name, column_name, data_type, data_length
  from SYS.DBA_TAB_COLS
  where owner = 'SCOTT'
```

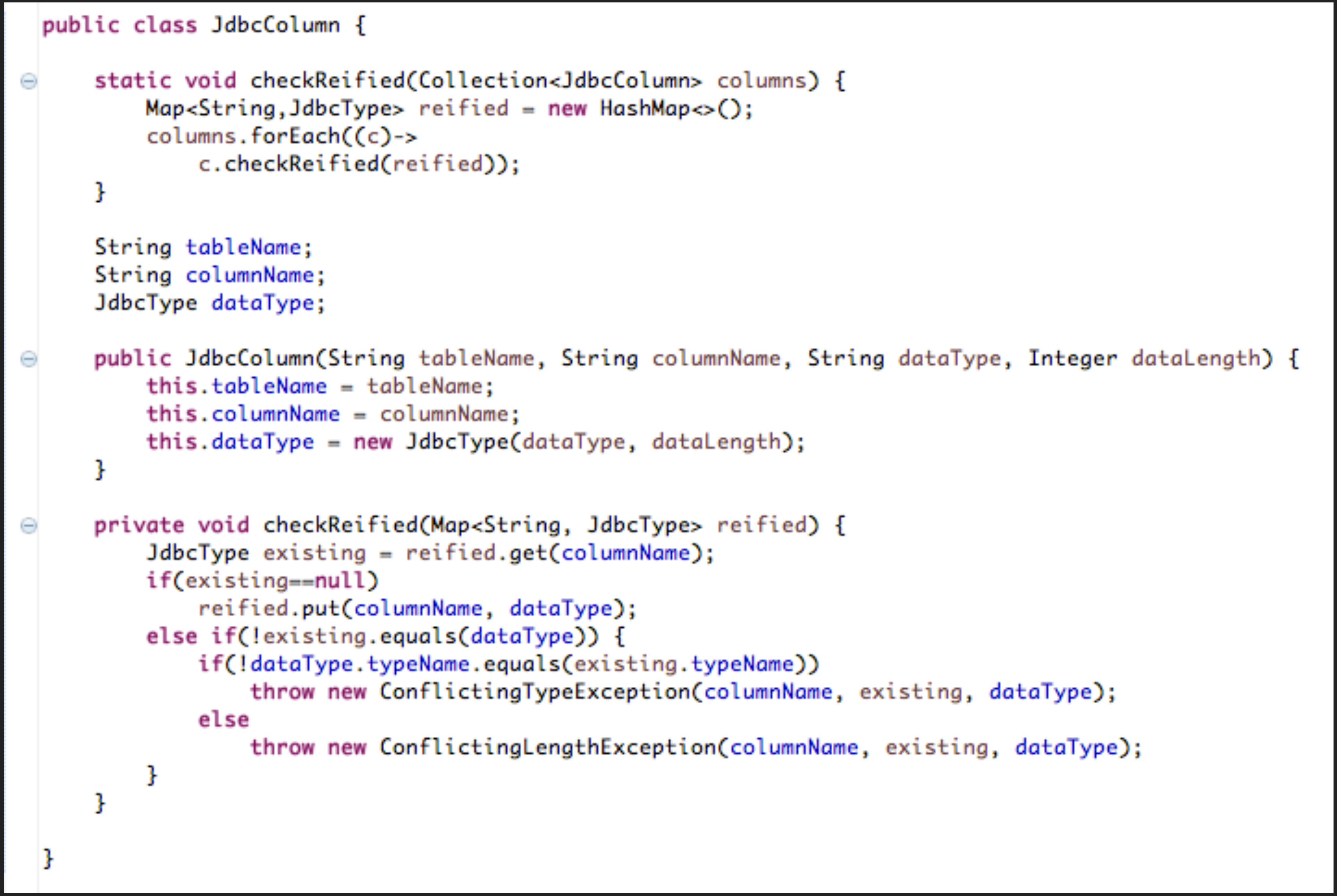
Résultat de requête ×

SQL | Toutes les lignes extraites : 36 en 0,044 secondes

	TABLE_NAME	COLUMN_NAME	DATA_TYPE	DATA_LENGTH
17	SAMPLE_DATASET_EVOLVE	XMLDATA	BLOB	4000
18	SAMPLE_DATASET_EVOLVE	SYS_NC_OID$	RAW	16
19	SALGRADE	HISAL	NUMBER	22
20	SALGRADE	LOSAL	NUMBER	22
21	SALGRADE	GRADE	NUMBER	22
22	EMP	DEPTNO	NUMBER	22
23	EMP	COMM	NUMBER	22
24	EMP	SAL	NUMBER	22
25	EMP	HIREDATE	DATE	7
26	EMP	MGR	NUMBER	22
27	EMP	JOB	VARCHAR2	9

- From there, verifying column types is also very simple

DETECTING SQL COLUMN TYPES INCONSISTENCIES

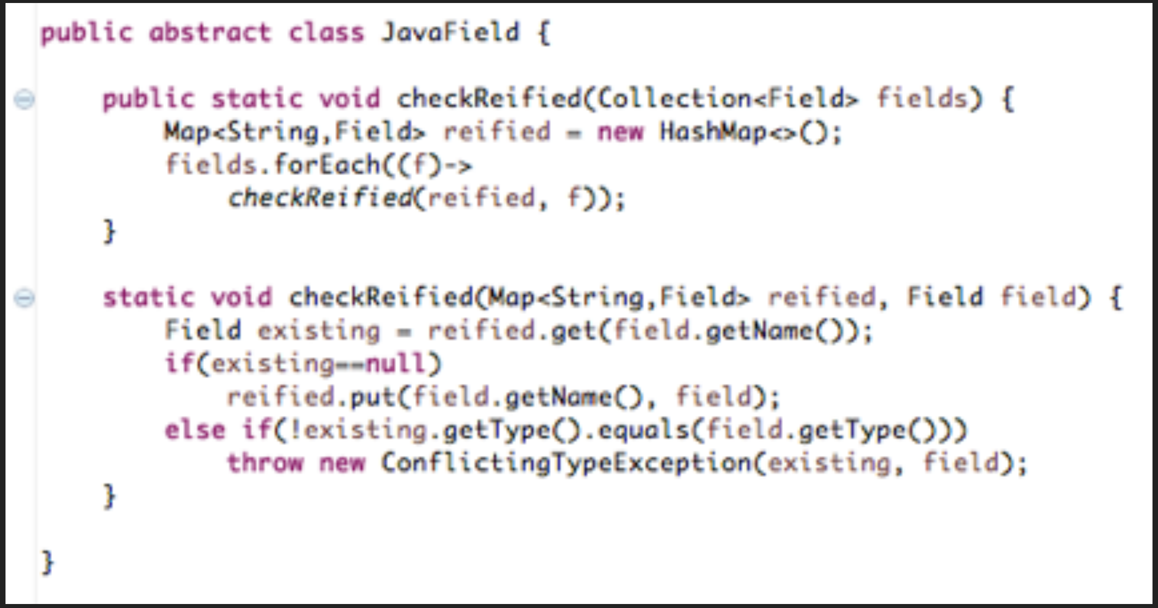


```java
public class JdbcColumn {

    static void checkReified(Collection<JdbcColumn> columns) {
        Map<String,JdbcType> reified = new HashMap<>();
        columns.forEach((c)->
            c.checkReified(reified));
    }

    String tableName;
    String columnName;
    JdbcType dataType;

    public JdbcColumn(String tableName, String columnName, String dataType, Integer dataLength) {
        this.tableName = tableName;
        this.columnName = columnName;
        this.dataType = new JdbcType(dataType, dataLength);
    }

    private void checkReified(Map<String, JdbcType> reified) {
        JdbcType existing = reified.get(columnName);
        if(existing==null)
            reified.put(columnName, dataType);
        else if(!existing.equals(dataType)) {
            if(!dataType.typeName.equals(existing.typeName))
                throw new ConflictingTypeException(columnName, existing, dataType);
            else
                throw new ConflictingLengthException(columnName, existing, dataType);
        }
    }

}
```

- Code is available at https://github.com/ericvergnaud/reifying-attributes

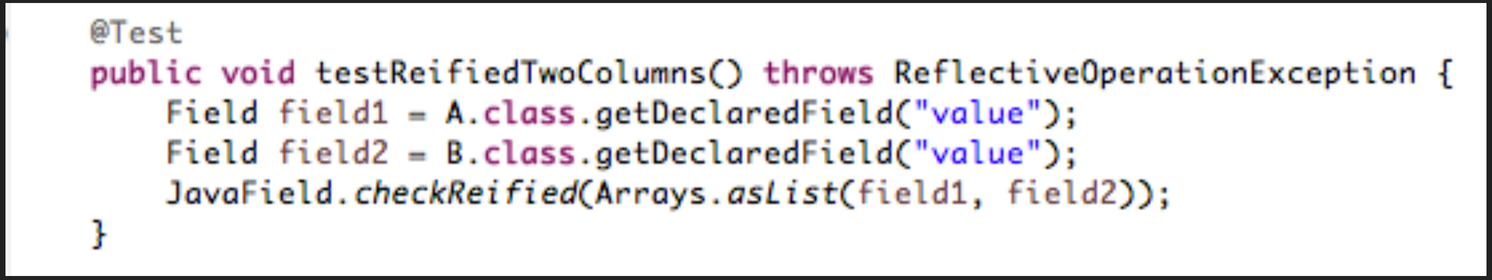
VERIFYING JAVA CLASSES USING REFLECTION

- Very easy to implement:



```
public abstract class JavaField {

    public static void checkReified(Collection<Field> fields) {
        Map<String,Field> reified = new HashMap<>();
        fields.forEach((f)->
            checkReified(reified, f));
    }

    static void checkReified(Map<String,Field> reified, Field field) {
        Field existing = reified.get(field.getName());
        if(existing==null)
            reified.put(field.getName(), field);
        else if(!existing.getType().equals(field.getType()))
            throw new ConflictingTypeException(existing, field);
    }

}
```

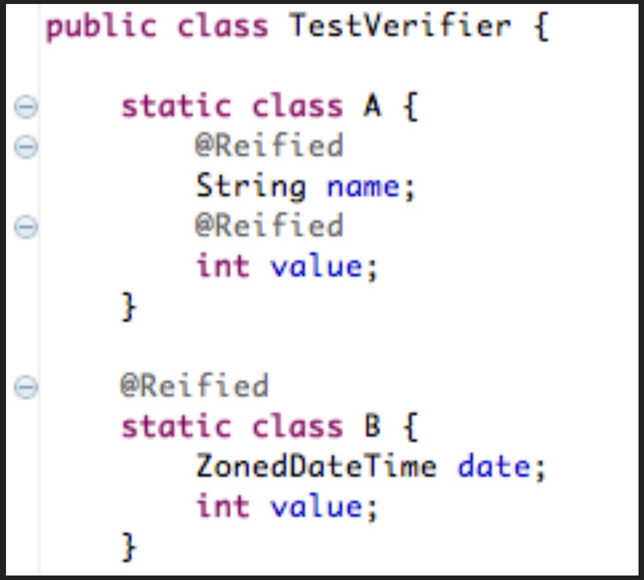
- But doesn't tell us which classes and fields to consider/ignore, very tedious



```
@Test
public void testReifiedTwoColumns() throws ReflectiveOperationException {
    Field field1 = A.class.getDeclaredField("value");
    Field field2 = B.class.getDeclaredField("value");
    JavaField.checkReified(Arrays.asList(field1, field2));
}
```

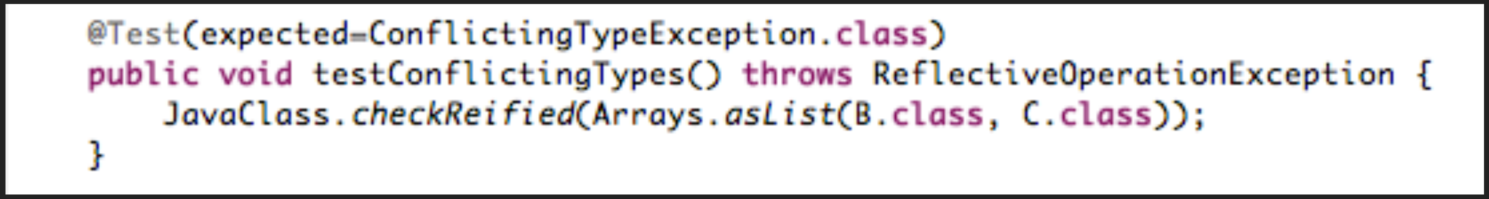
VERIFYING JAVA CLASSES USING ANNOTATIONS

- Much better:



```java
public class TestVerifier {

    static class A {
        @Reified
        String name;
        @Reified
        int value;
    }

    @Reified
    static class B {
        ZonedDateTime date;
        int value;
    }
```

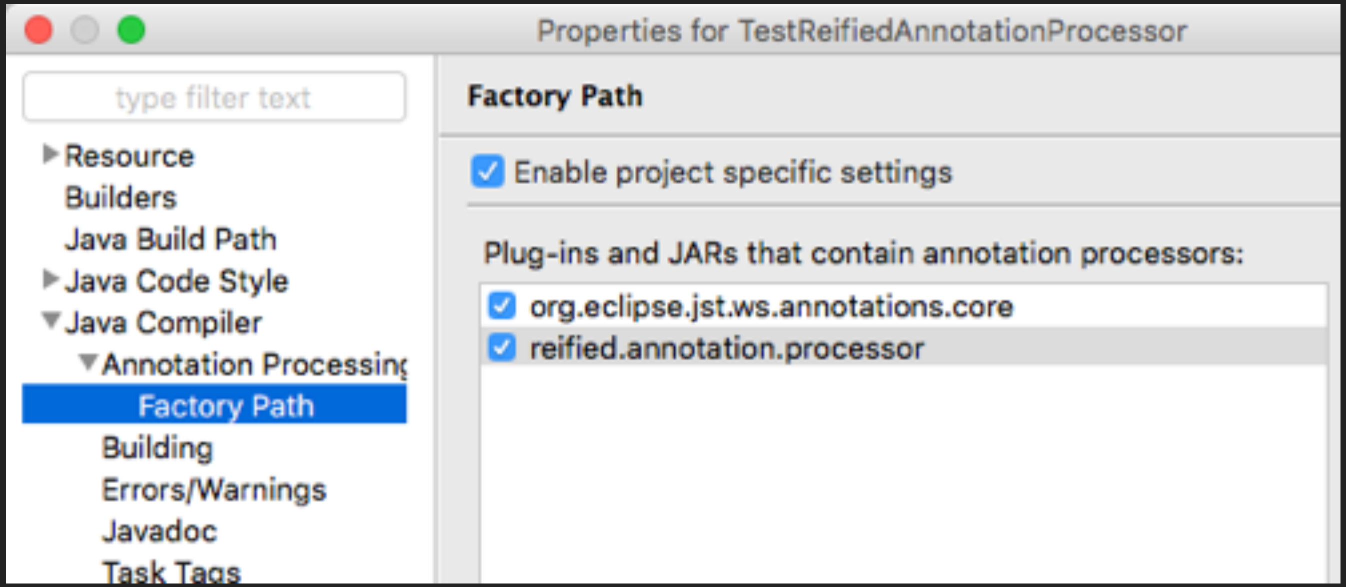
- But we still need to specify which classes to verify at build time, still tedious and error-prone



```java
@Test(expected=ConflictingTypeException.class)
public void testConflictingTypes() throws ReflectiveOperationException {
    JavaClass.checkReified(Arrays.asList(B.class, C.class));
}
```

VERIFYING AT COMPILE TIME USING AN ANNOTATION PROCESSOR

- Activate the annotation processor:


Properties for TestReifiedAnnotationProcessor

type filter text

- Resource
- Builders
- Java Build Path
- Java Code Style
- Java Compiler
  - Annotation Processing
    - Factory Path
  - Building
  - Errors/Warnings
  - Javadoc
  - Task Tags

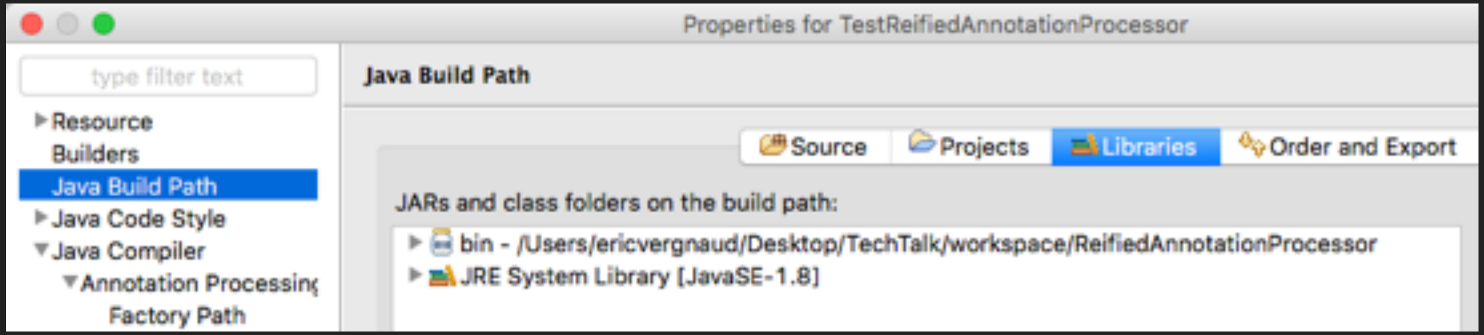
Factory Path

☑ Enable project specific settings

Plug-ins and JARs that contain annotation processors:

☑ org.eclipse.jst.ws.annotations.core
☑ reified.annotation.processor

- Also add the @Reified annotation class to the build path


Properties for TestReifiedAnnotationProcessor

type filter text

- Resource
- Builders
- Java Build Path
- Java Code Style
- Java Compiler
  - Annotation Processing
    - Factory Path

Java Build Path

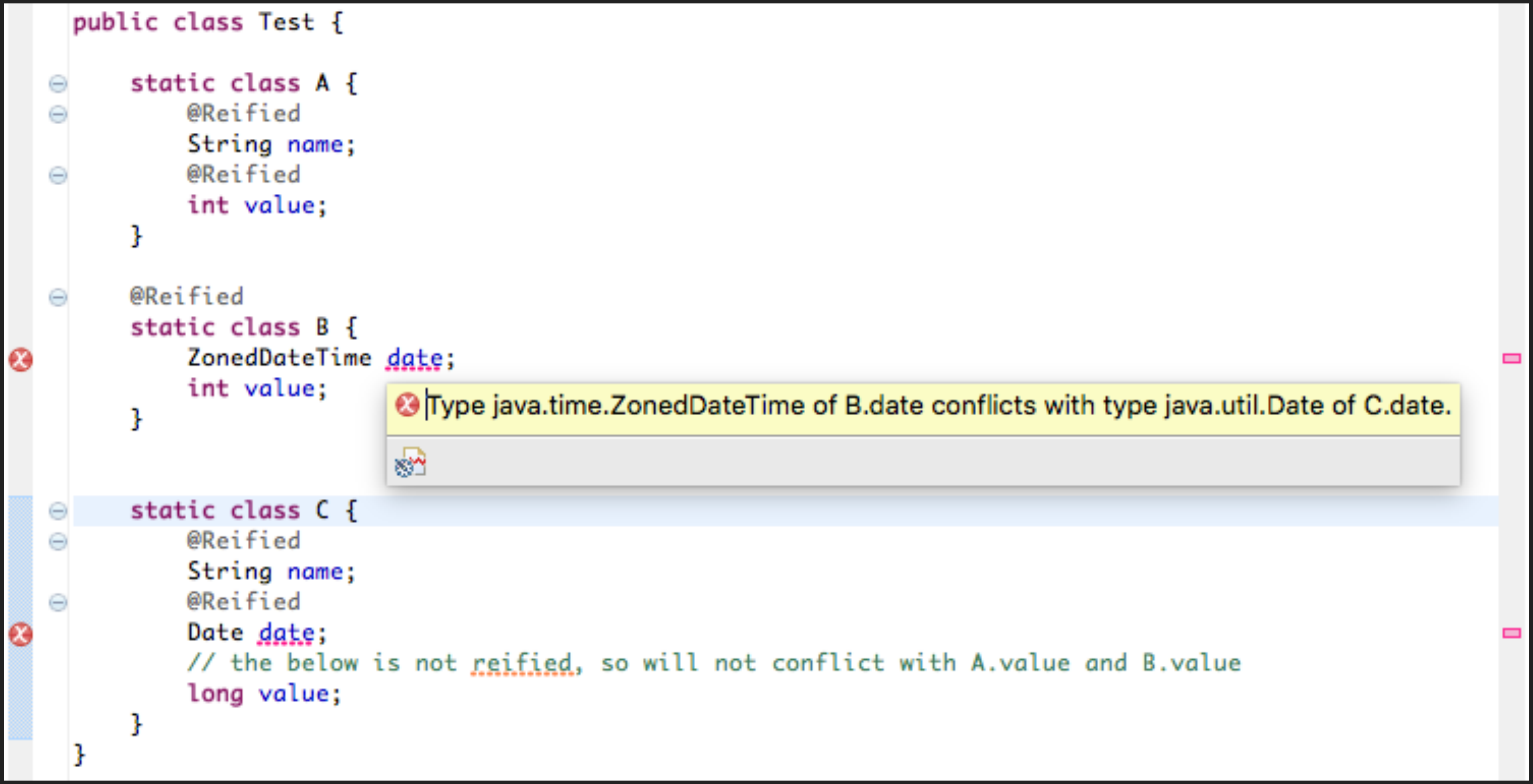
Source | Projects | Libraries | Order and Export

JARs and class folders on the build path:

- bin - /Users/ericvergnaud/Desktop/TechTalk/workspace/ReifiedAnnotationProcessor
- JRE System Library [JavaSE-1.8]

VERIFYING AT COMPILE TIME USING AN ANNOTATION PROCESSOR

- Here is how it looks like:



```
public class Test {

    static class A {
        @Reified
        String name;
        @Reified
        int value;
    }

    @Reified
    static class B {
        ZonedDateTime date;
        int value;
    }


    static class C {
        @Reified
        String name;
        @Reified
        Date date;
        // the below is not reified, so will not conflict with A.value and B.value
        long value;
    }
}
```

Type java.time.ZonedDateTime of B.date conflicts with type java.util.Date of C.date.

- Code is available at https://github.com/ericvergnaud/reifying-attributes

DISCUSSION