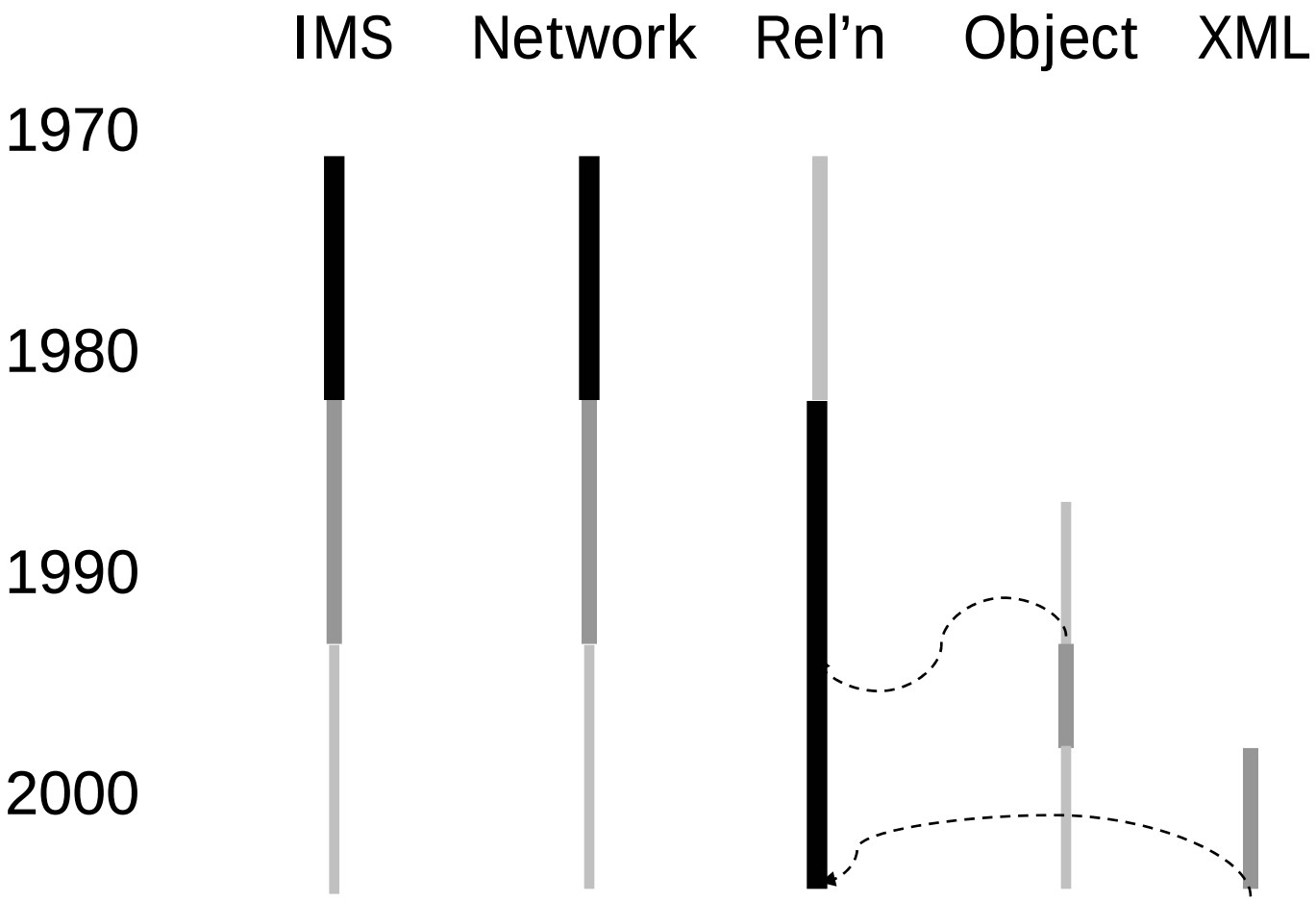
CS331: Advanced Database Systems: Object Databases

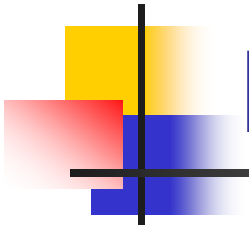
Norman Paton

The University of Manchester

norm@cs.man.ac.uk

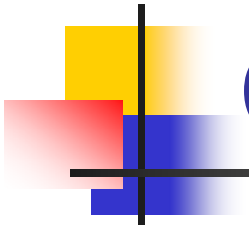
Data Model History





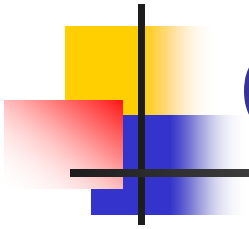
Relational Model Weaknesses

- Data model:
  - A single bulk data type (relation).
  - No direct support for:
    - Hierarchies (part-of or is-a).
    - Advanced data types (spatial, multimedia...).
- Programming:
  - Impedance mismatches due to query language emphasis.
  - Challenging to make existing application data persist.



Object Databases

- Motivation: to overcome weaknesses of relational approach:
  - Richer data models.
  - Closer integration with programming languages.
- Kinds of object database:
  - Object relational (Oracle, DB2, PostgreSQL).
  - Semantic data model (Jasmine).
  - Programming language centred (Objectivity, FastObjects, Versant, ObjectStore).

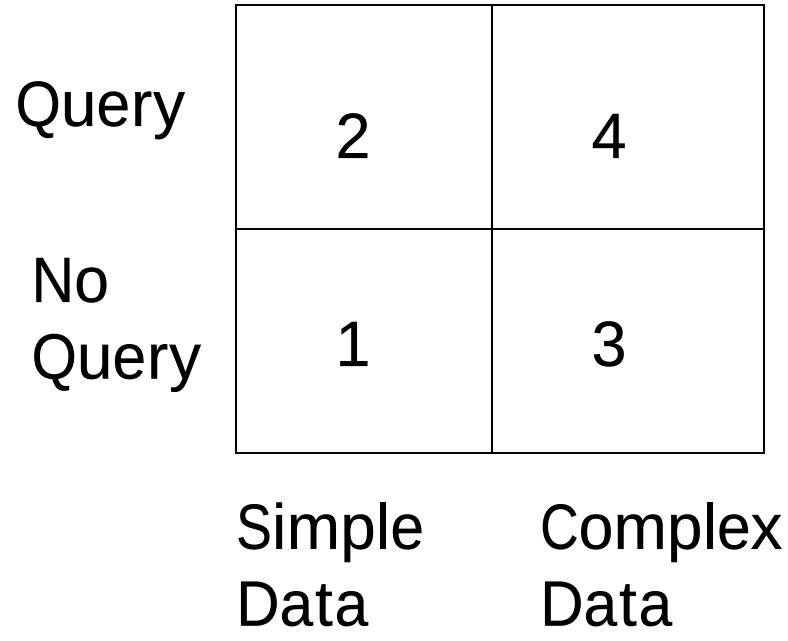


Object Database Standards

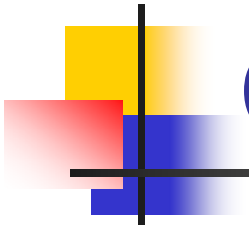
- SQL-99:
  - Mainstream SQL standard.
  - Oracle, DB2, Informix.
- Object data management group (ODMG):
  - Outgoing mainstream object database standard.
  - Objectivity, FastObjects, Versant.
- Java data objects (JDO):
  - Java-specific object-based database access.
  - Poet, Versant, ObjectStore, SolarMetric, ...

Application Emphasis

- SQL-99:
  - 2, 4.
- ODMG:
  - 3, 4.
  - But products support for queries is limited.
- JDO:
  - 3, 4.
  - But query language limited compared with SQL.

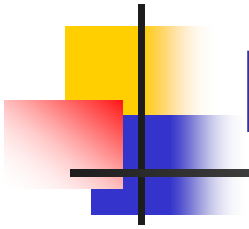


[Stonebraker 99]



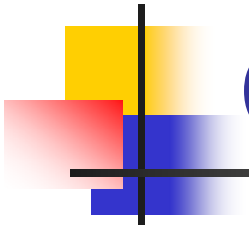
Object Relational Databases

- These add to the relational model:
  - Object types.
  - Nested tables.
  - References.
  - Inheritance.
  - Methods.
  - Abstract data types.
- The SQL-99 standard covers all of the above; in what follows, examples are from Oracle 9*i*.



Relational Model and Types

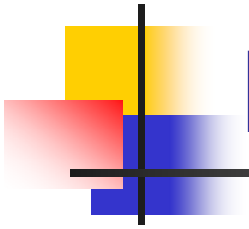
- Data type completeness: each type constructor can be applied uniformly to types in the type system.
- In the basic relational model:
  - There is only one type constructor (i.e. relation).
  - That type constructor cannot be applied to itself.
- Incorporating data type completeness to the relational model gives nested relations.
- In addition, the type *relation* is essentially:
  - Bag < Tuple >.
- Separating out these type constructors provides further flexibility, such as tuple-valued attributes.



Object Types in Oracle

- An object type is a user-defined data type, somewhat analogous to a class in object-oriented programming.
- Types can be arranged in hierarchies, and instances of types can be referenced.

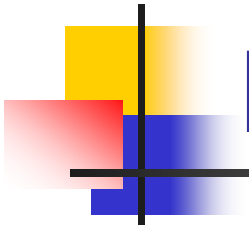
```
create type visit_type as object (
  name      varchar(20), /* the station */
  thetime   number
);
```



Nested Relations

- Nested relations involve the storage of one relation as an attribute of another.

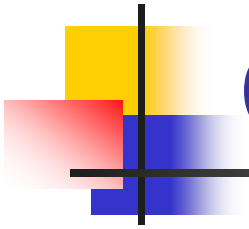
```
create type visit_tab_type as table of visit_type;

create table train (
    t#        varchar(10)     not null,
    type      char(1)         not null,
    visits    visit_tab_type,
primary key (t#))
nested table visits store as visits_tab;
```



Populating Nested Tables

- The name of the type can be used as a constructor for values of the type.

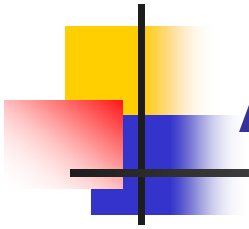
```
update train
set visits =
  visit_tab_type(
    visit_type('Edinburgh',950),
    visit_type('Aberdeen',720))
where t# = '22403101'
```



Querying Nested Tables

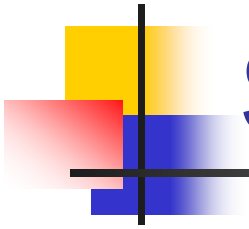
- Query operations such as unnesting allow access to the contents of a nested table.
- The following query retrieves details of the trains that visit *Inverness*.

```
select *
from train t, table(t.visits) v
where v.name = ‘Inverness’
```



Abstract Data Types

- Abstract data types allow new primitive types to be added to a DBMS (a.k.a. data blades, cartridges).
- These primitive types can be defined by (skilled) users or vendors.
- Oracle-supplied cartridges include:
  - Time.
  - Text.
  - Image.
  - Spatial.
  - Video.

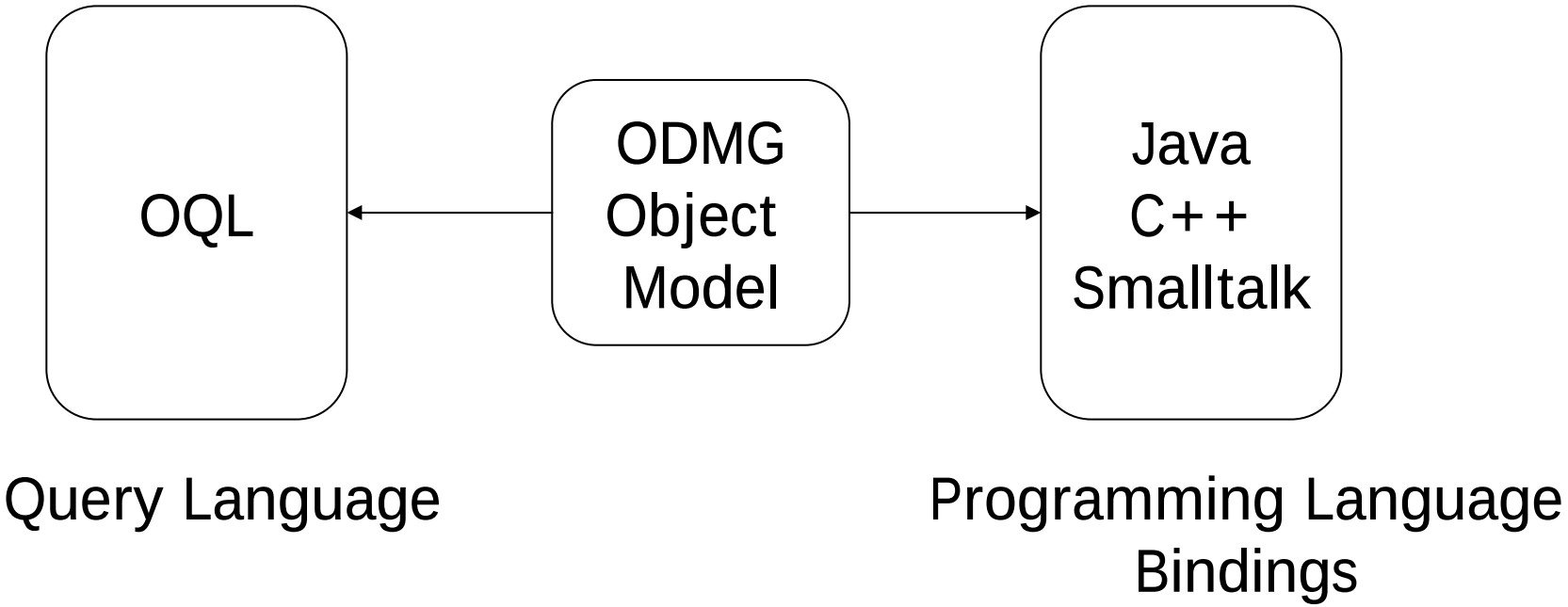


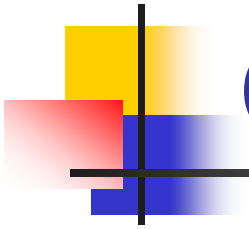
Summary on Object Relational

- Object relational features:
  - Are supported by all the major vendors.
  - Are not consistently in line with SQL-99.
  - Are used only periodically.
- Abstract Data Types/Data blades make a more fundamental differences to the capabilities of a relational DBMS than object model extensions, and are quite widely used.

ODMG Object Model

- The ODMG Object Model was the first standard for non-evolutionary object databases.

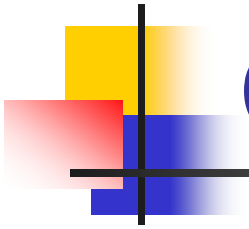




ODMG Object Model

- The object model provides a programming language-independent notation for object modelling.

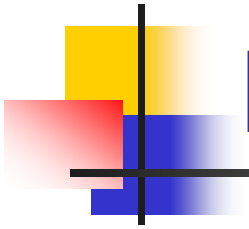
```
class main : station
{ ...
  relationship Set<district> district_stations
    inverse district::main_station;
  ...
}

class district : station
{ ...
  relationship main main_station
    inverse main::district_stations;
}
```



OQL

- OQL provides one way of accessing data in an ODMG database.
- The following query, the result of which is of type *Bag<district>*, retrieves the district stations associated with the main station with name *Manchester*.

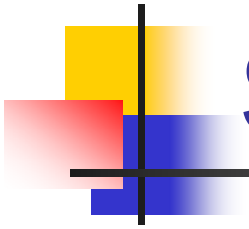
```
select d
from district d
where d.main_station.name = “Manchester”
```



Language Bindings

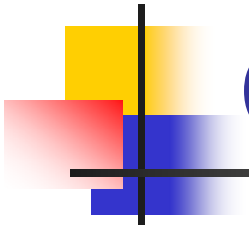
- Language bindings map database classes to programming language classes, and provide associated utility functionalities.

```
import com.poet.odmg.*;
...
Database db = new Database();
db.open("TrainsDB", Database.OPEN_READ_WRITE);

Transaction txn = new Transaction();
txn.begin();
Train theTrain = new Train(...);
db.bind(myObject, theTrain.getTno());
txn.commit();
```



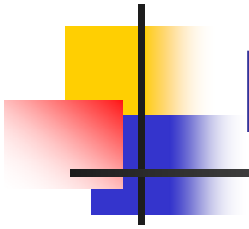
Summary on ODMG

- The ODMG model and language bindings are supported by several object database vendors.
- There is fairly limited support in products for OQL – object databases tend to be programming-language focused.
- Although ODMG is programming-language neutral, programming-language specific proposals, such as JDO, seem to be taking over.



Object Databases Summary

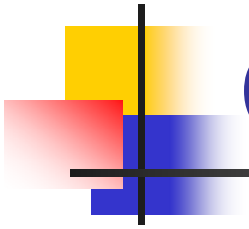
- Object data modelling supports application requirements, especially in scientific and engineering domains.
- The history and evolution of object databases has been complex ... and the future is not certain.
- Object data management capabilities are increasingly supported as part of existing environments rather than as distinct systems in their own right.



Further Reading

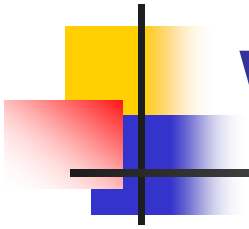
- M. Stonebraker, P. Browne, Object-Relational Databases, 2nd Edition, Morgan-Kaufmann, 1999.
- W. Gietz, Oracle 9*i* Application Developers Guide – Object Relational Features, 2002 (Chapter 1: Introduction; Chapter 2: Basic Components of Oracle Objects).
- R. Cattell, The Object Data Standard: ODMG 3.0, Morgan-Kaufmann, 2000.

Java Data Objects



Overview of JDO

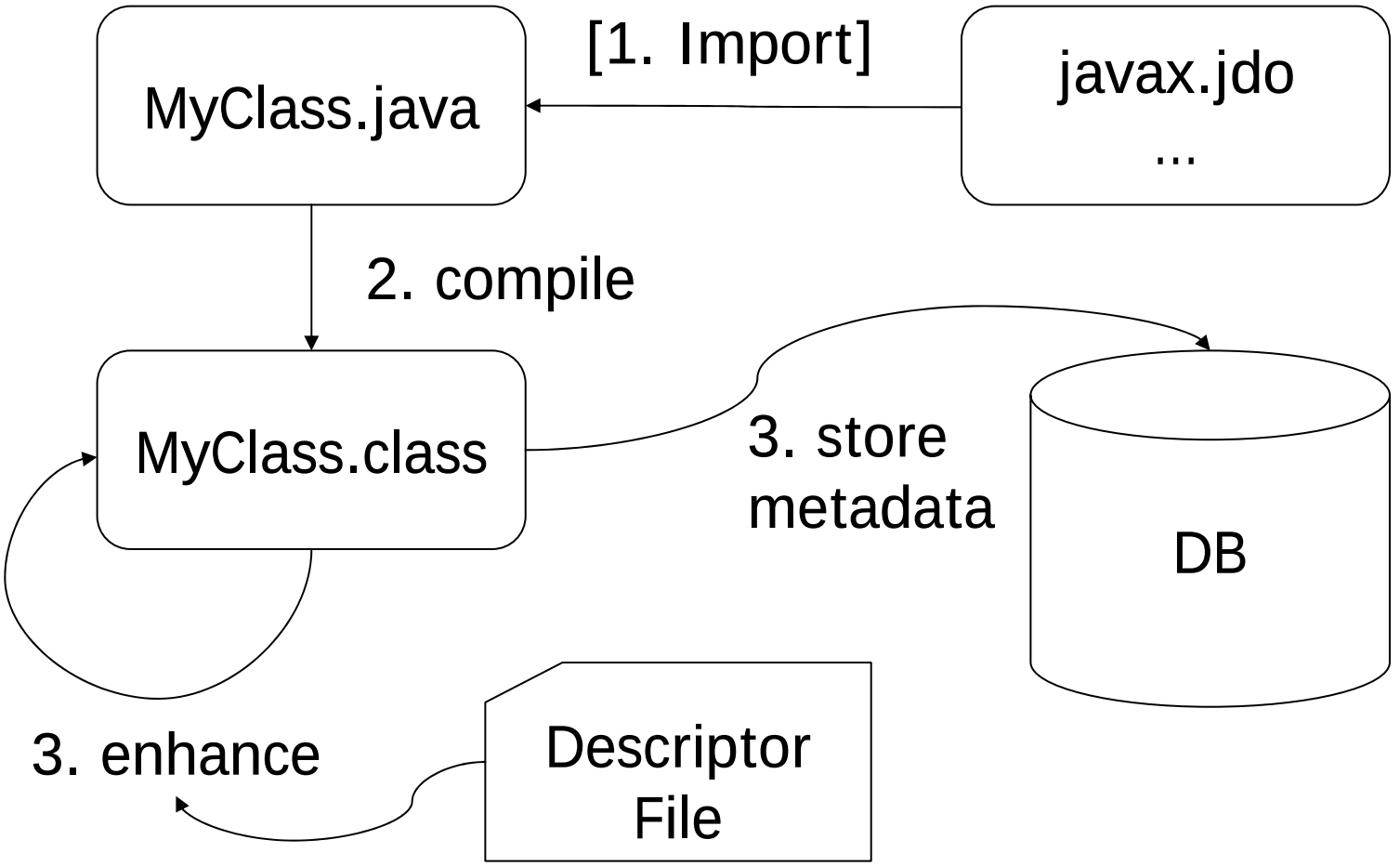
- JDO supports:
  - The storage of Java objects in a database:
    - An object store, or
    - A relational store.
  - Access to the stored Java objects:
    - Through queries embedded in Java.
    - By iterating through the instances of a class.
    - By navigating between objects in the database.
- Database objects are created and manipulated in the same way as main memory objects.

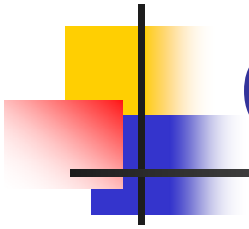


Writing JDO Applications

- From scratch:
  - Java classes are implemented as normal, adhering to certain restrictions on the types that can be stored in the database.
  - Java methods can use JDO functionality to access and manipulate database data.
  - A description file describes what classes are database classes, etc.
- Based on existing Java classes:
  - Java classes are checked to see if they adhere to certain restrictions the types that can be stored in the database.
  - Methods won't initially use JDO functionality, but top-level programs must be adapted to access database data.
  - A description file describes what classes are database classes, etc.

Registering a Persistent Class

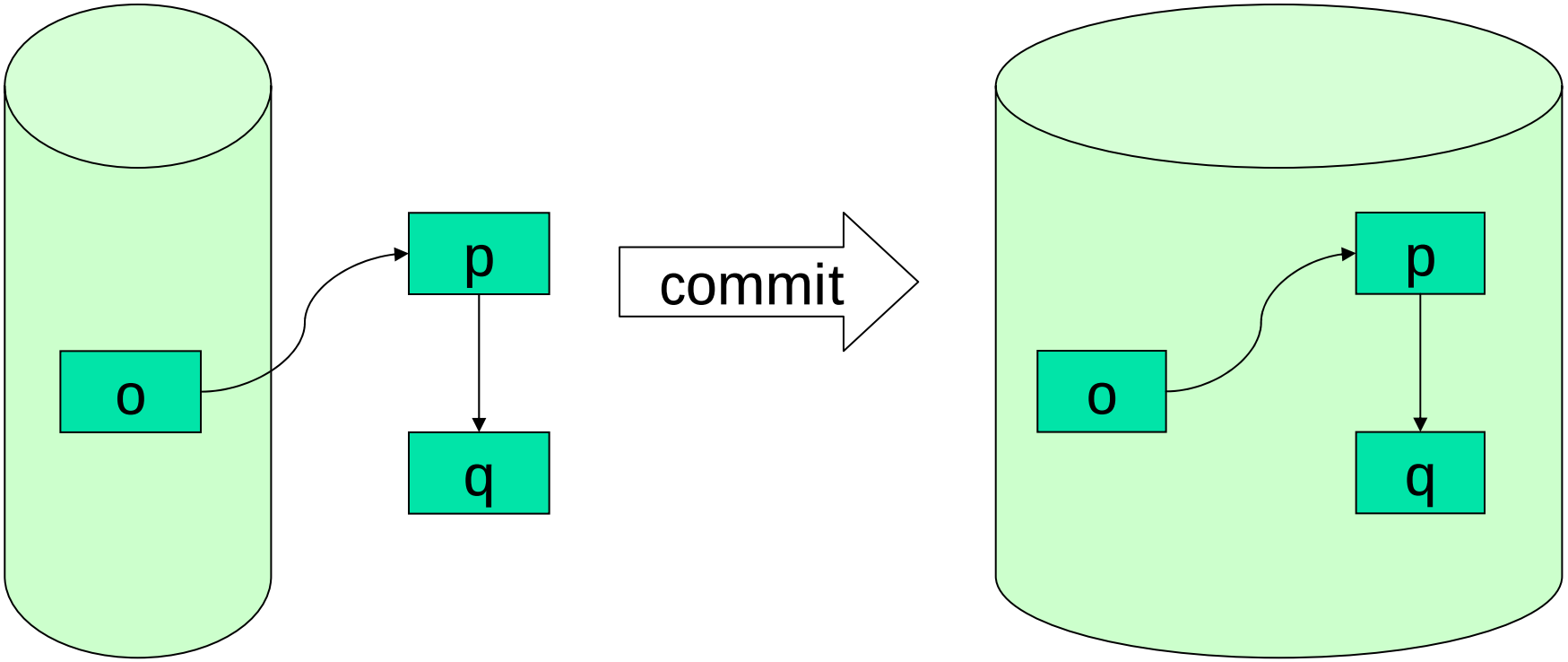


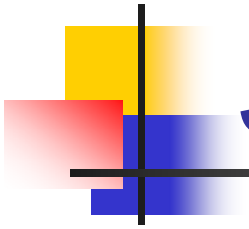


Object Database Notions

- Extent:
  - The collection of instances of a class.
- Orthogonal Persistence:
  - The length of time that a data item can exist is independent of its type.
- Persistence by reachability:
  - An object can be made to persist either explicitly, or by being referenced by a persistent object.

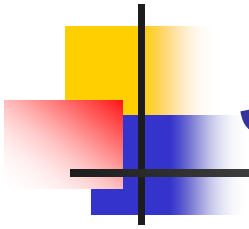
Persistence by Reachability





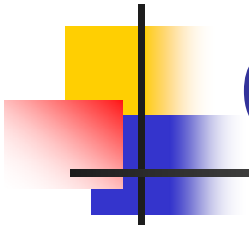
JDO Data Model

- JDO data model ≅ the Java type system.
- Features:
  - Classes.
  - Inheritance.
  - Members.
  - Collections.
- Restrictions:
  - Only certain collection types can be stored in the database.
  - Only serialisable library types can be stored in databases.
- Additions:
  - Keys.
  - Collection types.
  - Vendor-specific features (e.g. indexes, clustering).



JDO Supported Types

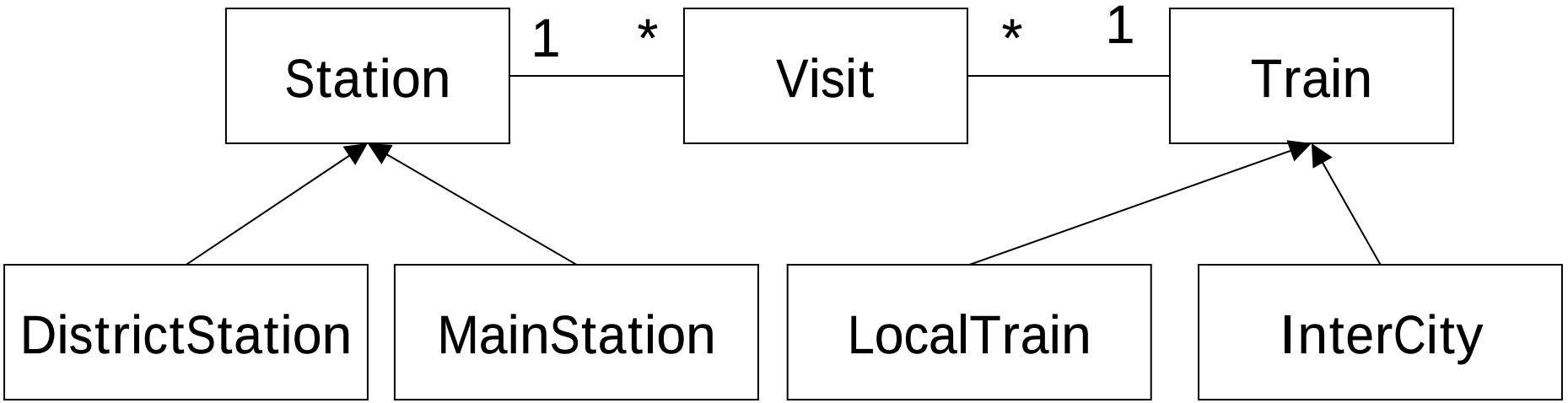
- Primitives:
  - Boolean, byte, short, char, int, long, float, double.
- java.lang:
  - Wrappers for primitives, plus String, Number, Object.
- java.util:
  - Locale, Date.
  - Collections.
- java.math:
  - BigInteger, BigDecimal.



Collections

- JDO includes the following Collection classes:
  - Set:
    - HashSet, Hashtable, TreeSet.
  - List:
    - ArrayList, LinkedList, Vector.
  - Map:
    - HashMap, TreeMap.
- Arrays:
  - JDO supports one dimensional arrays.

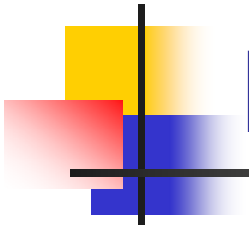
Trains Database Schema



See handout/tutorial for the JDO schema and example programs.

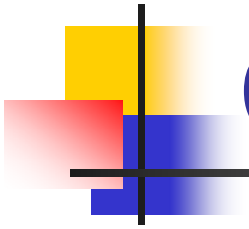
Train Class (nothing to see)

```
class Train {
  protected int tno ;
  protected Station source ;
  protected Station dest ;
  protected Vector route ;
  protected char type ;

  // Mandatory 0-ary Constructor
  public Train ( ) {}

  public Train (int the_no, Station the_source,
    Station the_dest, Vector the_route)
  { tno = the_no ; source = the_source ;
    dest = the_dest ; route = the_route ; type = 't' ;
  }
  …
}
```



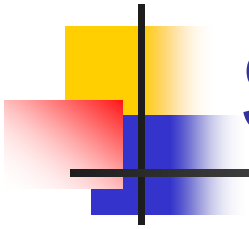
Minimal Descriptor File

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="">
    <class name="DistrictStation">
    <class name="InterCity">
    <class name="LocalTrain">
    <class name="MainStation">
    <class name="Station">
    <class name="Train">
    <class name="Visit">
    </class>
  </package>
</jdo>
```



Creating a Persistent Object

1. Connect to the database.
2. Open a transaction.
3. Create the Java object.
4. Store the Java object.
5. Commit the transaction.



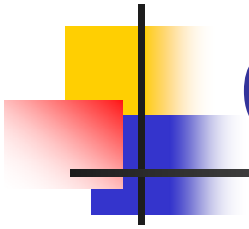
Step 1: Connect

```
import javax.jdo.* ;
import com.poet.jdo.*;
…
java.util.Properties pmfProps = new java.util.Properties();
pmfProps.put(
  "javax.jdo.PersistenceManagerFactoryClass",
  "com.poet.jdo.PersistenceManagerFactories" );
pmfProps.put(
  "javax.jdo.option.ConnectionURL",
  "fastobjects://LOCAL/MyBase" );
PersistenceManagerFactory pmf =
  JDOHelper.getPersistenceManagerFactory( pmfProps );

PersistenceManager pm = pmf.getPersistenceManager();
```

Steps 2-5: Do the business

```
Transaction txn = pm.currentTransaction();
txn.begin();
…
Station source = …;
Station dest = …;
LocalTrain tr = …;
…
tr = new Train(tno, source, dest, new Vector ());
pm.makePersistent (tr)
…
txn.commit();
```

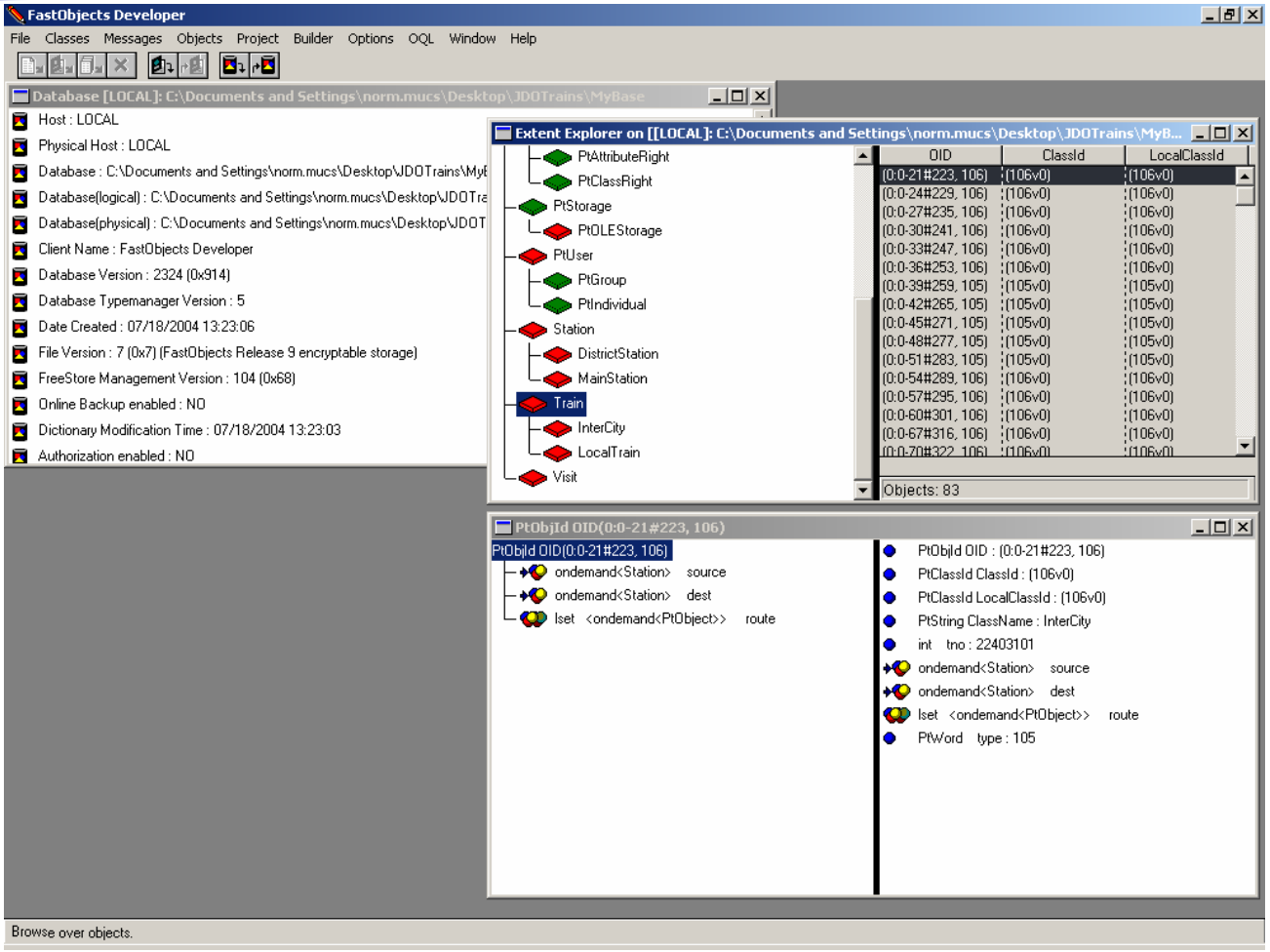
This will also ensure that *source* and *dest* persist.



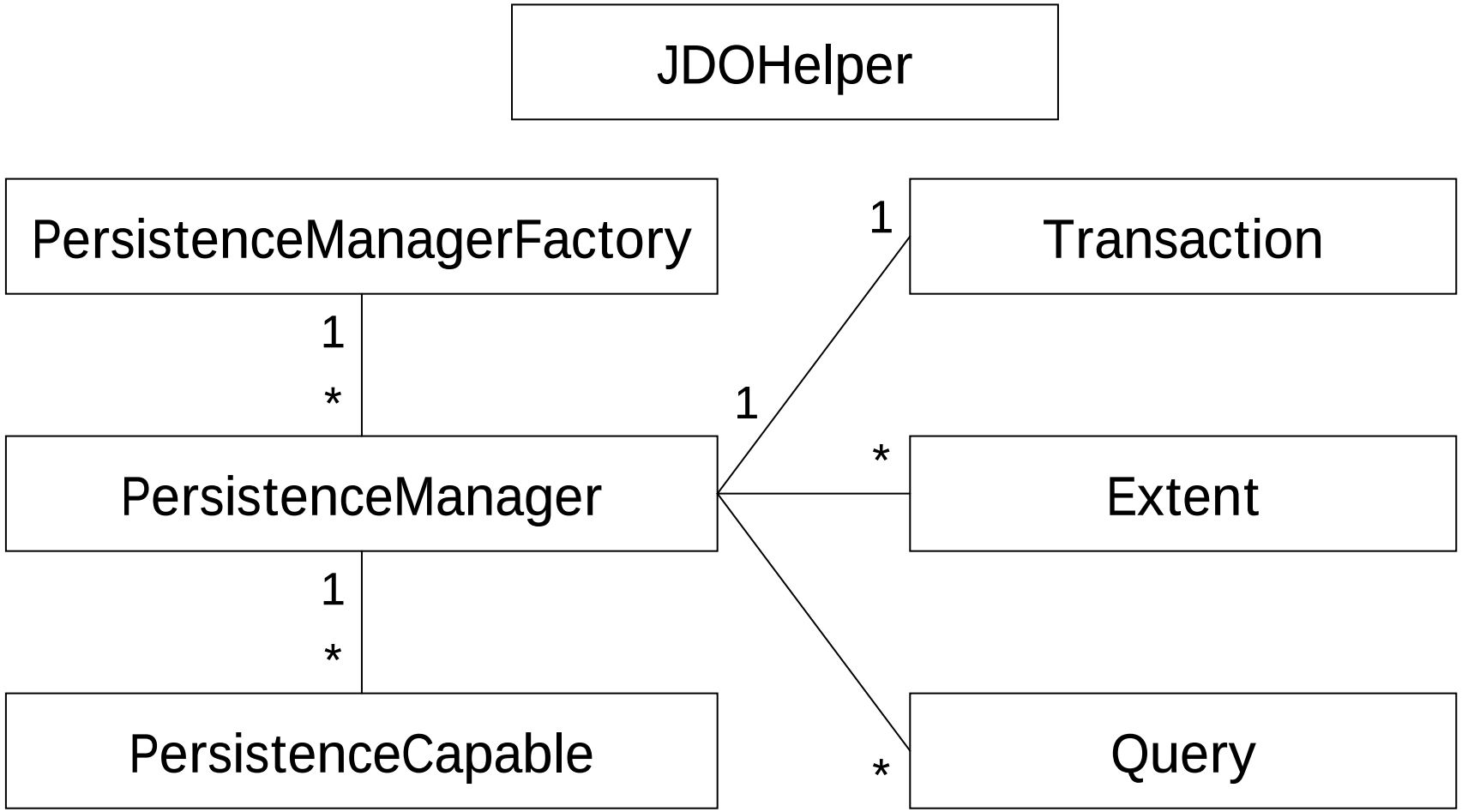
Compiling in FastObjects

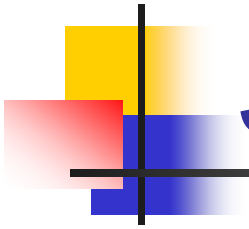
```
>javac *.java
Note: Trains.java uses or overrides a deprecated API.
Note: Recompile with -deprecation for details.

>ptj -enhance -update -schema MySchema
FastObjects SDK Toolkit, Version 9.5.12.98.
Copyright (C) 1996-2004 POET Software Corporation
Info:  Read metadata 'C:\JDOTrains\package.jdo'
Info:  Enhanced class '.\DistrictStation.class'
Info:  Enhanced class '.\Visit.class‘
…
Access to poet://LOCAL/MySchema failed
[the schema was not found (-2031)].
Create a new schema (y/n)? y
Info: Created schema: poet://LOCAL/MySchema
…
```

Viewing Objects in FastObjects



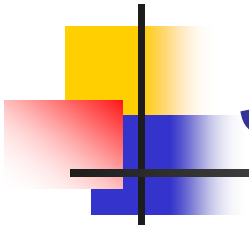
JDO Classes - 1





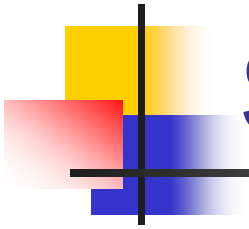
JDO Classes - 2

- *JDOHelper*: Static utility operations for creating *PersistenceManagerFactory* objects and enquiring about instances.
- *PersistenceManagerFactory*: A factory for creating *PersistenceManager* objects.
- *PersistenceManager*: Manages access to a collection of persistent objects (comparable to a *Connection* in JDBC).



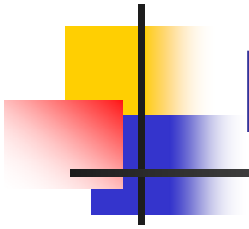
JDO Classes - 3

- *PersistenceCapable*: Interface implemented by database classes; normally added during enhancement.
- *Transaction*: Supports grouping of operations for *commit*/ *rollback*.
- *Extent*: Enables access to the instances of a class (and its subclasses).
- *Query*: Provides declarative access to persistent objects.



Summary

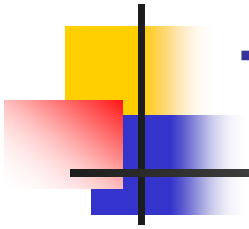
- JDO provides (largely) orthogonal persistence for Java.
- JDO involves no extensions to Java syntax or compilers.
- JDO can be used with specialist object managers or relational stores.
- JDO is now widely supported, both by object database and other vendors.
- JDO is a Java Community Process Standard (JSR-12).



Further Reading

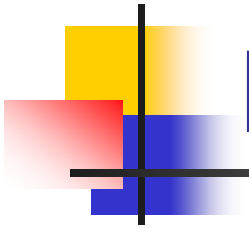
- D. Jordan, C. Jones, Java Data Objects, O'Reilly, 2003.
- JDOCentral: many articles including tutorials (http://www.jdocentral.com/).
- A particularly useful tutorial is:
  - http://www.solarmetric.com/Software/Documentation/2.3.0/jdo-overview.html

Accessing JDO Databases



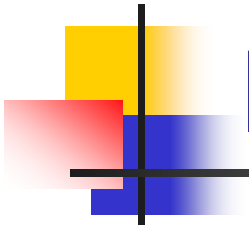
Topics

- Establishing connections.
- Accessing objects via extents.
- Accessing objects via navigation.
- Updates.
- Descriptor files.



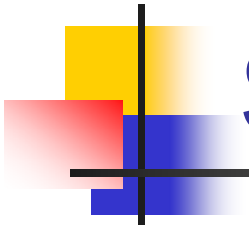
PersistenceManager Class

- A persistence manager can be seen as coordinating access with the database.
- A *PersistenceManagerFactory* object represents a database to a program.
- A *PersistenceManager* object represents a connection to a database from a program.



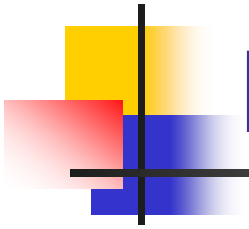
Database Properties

- The following are among the properties that can be set for a *PersistenceManagerFactory*:
  - The name of the underlying persistence manager.
  - The name of the database that is to be accessed using that persistence manager.
  - The username and password to be used to access the database.
  - Various features of cache and transaction management.
- These properties are set using a Properties object that provides key $\rightarrow$ value pairs.



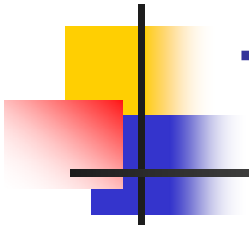
Setting Property Values

```
java.util.Properties pmfProps = new java.util.Properties();
pmfProps.put(
  "javax.jdo.PersistenceManagerFactoryClass",
  "com.poet.jdo.PersistenceManagerFactories" );
pmfProps.put(
  "javax.jdo.option.ConnectionURL",
  "fastobjects://LOCAL/MyBase" );
pmfProps.put(
  "javax.jdo.option.ConnectionUserName",
  “norm" );
pmfProps.put(
  "javax.jdo.option.ConnectionPassword",
  “notMyRealPassword" );
PersistenceManagerFactory pmf =
  JDOHelper.getPersistenceManagerFactory( pmfProps );
```



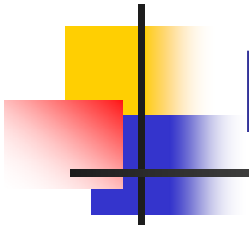
PersistenceManager Class

- The PersistenceManager Class has many operations, including:
  - `Transaction currentTransaction();`
  - `void deletePersistent(Object obj);`
  - `void makePersistent(Object obj);`
  - `void getExtent(Class c, boolean subclasses);`
  - `void newQuery();`
- As such, it is central to the explicit interaction of a client with a database.



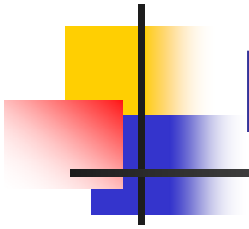
Transactions

- All updates to a database must take place as part of a transaction.
- The *Transaction* class supports operations such as:
  - void begin();
  - void commit();
  - void rollback();
  - boolean isActive();



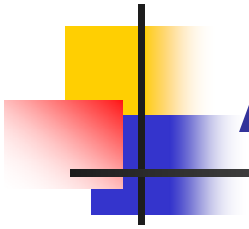
Extents

- An extent is the collection of instances in a class.
- Note that:
  - In relational databases, a table definition associates the definition of a type with a stored collection of its instances.
  - In object-oriented programming, there is not always a direct way of iterating over the instances of a class.



Extents in Action

```
// Print information about all trains
PersistenceManager pm = …

Transaction txn = pm.currentTransaction();
txn.begin();

Extent trainExtent = pm.getExtent( Train.class, true );
Iterator iter = trainExtent.iterator();
while ( iter.hasNext() )
{
  Train tr = (Train) iter.next();
  tr.print();
}

txn.commit()
```

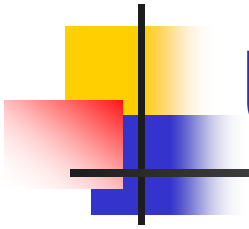


Access by Navigation - 1

- Once a persistent object has been retrieved, related persistent objects can be retrieved by navigating.
- As would be expected:
  - Single-valued relationships are followed using “.” notation.
  - Multiple-valued relationships are represented and accessed using collections.
  - No additional syntax is required in either case.
- Note that navigating through relationships is comparable to relational joins.

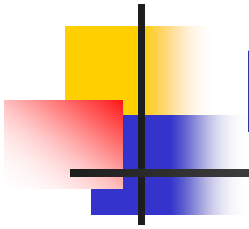
Access by Navigation - 2

```
class Train {
  …
  public void print ()
  {
    System.out.print ("Train number : " + tno + "\n" +
      "Source       : " + source.getName() + "\n" +
      "Destinantion : " + dest.getName() + "\n" +
      "Type         : " + this.getType() + "\n" ) ;

    Iterator iter = route.iterator();
    while ( iter.hasNext() )
    {
      Visit the_route = (Visit) iter.next() ;
      the_route.print() ;
    }
  }
}
```



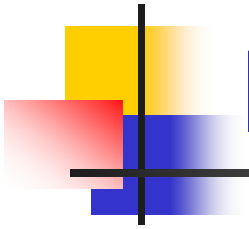
Updates

- The modification of members and of collections is the same as in Java.
- Deletion is more complex:
  - The notion of extent to some extent conflicts with persistence by reachability, as an explicitly made persistent object will continue to live in an extent even when no other object refers to it.
  - Thus while objects become persistent by reachability, there is no persistent garbage collection in JDO.



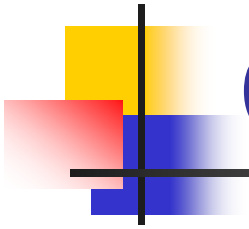
Deletion

- An object can be explicitly deleted from the data store using the following operations on *PersistenceManager*:
  - void deletePersistent(Object obj)
  - void deletePersistentAll(Object[] objs)
  - void deletePersistent(Collection objs)
- There is potential to create persistent dangling references by deleting an object, but not the references to it (yug).



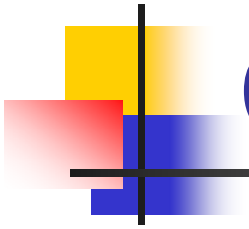
Deleting a Station

- Deletion thus needs to be designed based on application requirements.
- A possible approach for Station:
  - Disallow deletion of any station that is the source or destination of any train.
  - Allow deletion of a station to which there are visits, and delete the relevent visits.
- Writing a program that implements this behaviour is part of the tutorial.



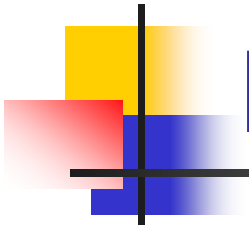
Configuration File - 1

- The JDO standard includes an XML DTD for a file that specifies properties of classes that cannot be captured in Java, such as:
  - `primary-key`: uniqueness constraint.
  - `embedded`: description of clustering.
  - `element-type`: typing for collections.



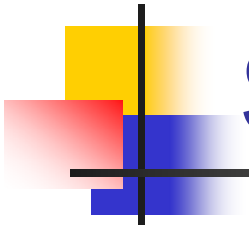
Configuration File - 2

- The configuration file provides scope for vendor extensions.
- FastObjects extensions include:
  - `alias`: for multiple-language access.
  - `index`: for more efficient querying.
  - `unicode`: to save space (`false` for ASCII).
  - `database`: name of database file.
  - `extent`: to save space (`false` for no extent).



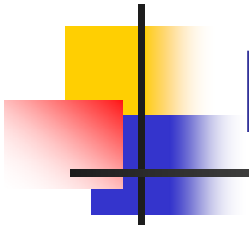
Example Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <extension vendor-name="FastObjects"
    key="database" value="MyBase"/>
  <extension vendor-name="FastObjects"
    key="schema" value="MySchema"/>
  <package name="">
    …
    <class name="Station">
          <field name="visits">
                <collection element-type="Visit"/>
          </field>
    </class>
  </package>
</jdo>
```



Summary:

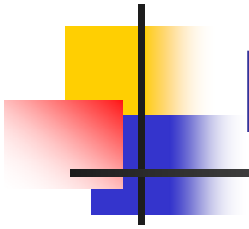
- There is not much to say about accessing databases in JDO ... which is the whole point!
- Access to objects is through extents, which support the Java iteration, or by navigation using standard Java syntax.
- Deletion of persistent objects is explicit ... and potentially tricky.
- Java class files don’t say everything you might want to say about a database schema, hence the need for configuration files.



Further Reading

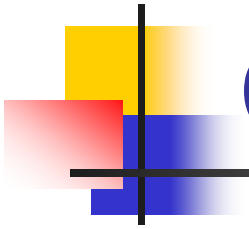
- Sources mentioned for the previous lecture apply.
- There are open-access versions of JDO if you want to try it, such as the Sun reference implementation:
  - http://access1.sun.com/jdo/

Querying JDO Databases using JDOQL



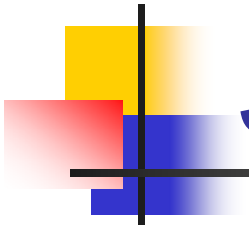
Role of Query Language

- Relational databases:
  - SQL supports:
    - Queries (DML).
    - Updates (DML).
    - Schema Manipulation (DDL).
  - SQL is central to program-based access from diverse programming languages.
- Object Databases:
  - Query languages have tended to support:
    - Queries.
    - ...but not updates or data manipulation.
  - Query languages are not mandatory for application programming.



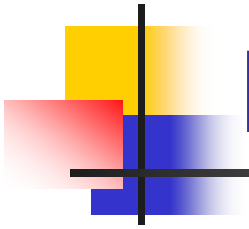
Object Query Languages

- SQL-99:
  - Extensions to SQL to support definition, creation and manipulation of objects in a relational setting.
  - Many extensions to original SQL, both structural and behavioural. Where will it all end?
  - Long lead time; somewhat patchy uptake.
- OQL:
  - Part of the Object Data Management Group (ODMG) object database standard.
  - Clean query expression, but no schema or data updates.
  - Use embedded or in interactive form.
  - Limited support from vendors.



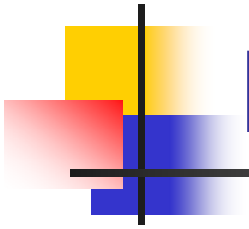
JDOQL

- JDOQL:
  - Is only for use embedded in Java programs.
  - Is thus seen as complementary to extents and navigation.
  - Pros/cons of embedded queries as against navigation?
- Scope:
  - Query only – no schema or data modification.
  - Principal model involves filtered access to extents, as an alternative to iteration.
  - Less expressive than OQL.



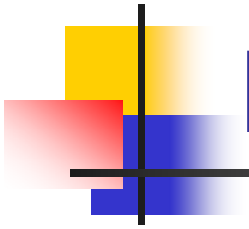
Example: Look for a Station

```
Extent ext = pm.getExtent(Station.class, true);

Query qry = pm.newQuery();
qry.declareParameters("String name");
qry.setFilter("this.name == name");
qry.setCandidates(ext);
Collection res = (Collection) qry.execute(argv[0]);

Iterator iter = res.iterator();
if (iter.hasNext())
{ // There should be exactly one
  Station foundStn = (Station) iter.next();
  foundStn.print();
} else {
  System.out.println("Station " + argv[0] + " not found");
}
```



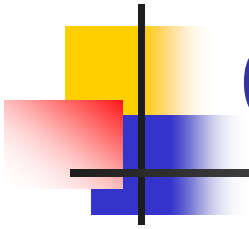
Features of Example – 1

- A query is represented as an object:
  - `Query qry = pm.newQuery();`
- A query returns a subset of a given collection:
  - `qry.setCandidates(ext);`
- A filter describes which of the values in the collection should feature in the result:
  - `qry.setFilter("this.name == name");`



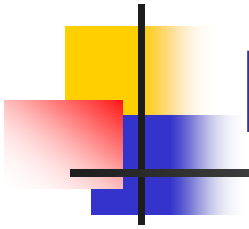
Features of Example - 2

- Queries can be passed parameters, which have to be declared:
  - `qry.declareParameters("String name");`
- Parameters can be set when a query is executed:
  - `Collection res = (Collection) qry.execute(argv[0]);`
- Query results are collections, and thus can be iterated over:
  - `Iterator iter = res.iterator();`



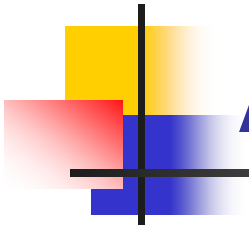
Class Query

- Queries can be constructed using the following operations on *PersistenceManager*:
  - `Query newQuery();`
  - `Query newQuery(Extent candidates);`
  - `Query newQuery(Extent candidates, String filter);`
  - `Query newQuery(Class candidateClass, Collection candidates);`
  - `Query newQuery(Class candidateClass, Collection candidates, String filter);`
- The `Collection` or `Extent` should contain persistent instances from the same *PersistenceManager* as the `Query`.



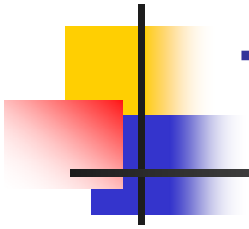
Formal Query Parameters

- Parameters are declared using the following operation on *Query*:
  - `void declareParameters(String parameters);`
- Parameters are comma separated, and can have any type that can be stored in a JDO database. Example:
  - `qry.declareParameters("String dest, Train trn");`



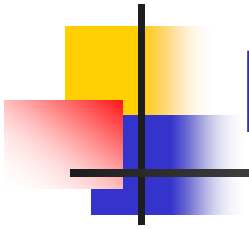
Actual Parameters

- Parameters values can be passed using the following operations on *Query*:
  - `execute`:
    - `Object execute();`
    - `Object execute(Object par1);`
    - `Object execute(Object par1, Object par2);`
    - `Object execute(Object par1, Object par2, Object par3);`
  - `executeWithMap`, which contains parameter name → value pairs.
    - `Object executeWithMap(HashMap map);`
- Primitives are wrapped (e.g. *int* → *Integer*)



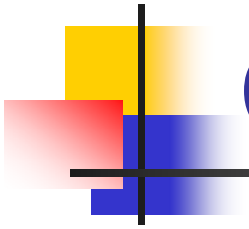
Types in Queries

- The type namespace for queries automatically includes:
  - The candidate class, and other classes in the same package.
  - Classes in `java.lang.*`.
- Other classes can be imported as in:
  - `qry.declareImports("import java.util.Date");`



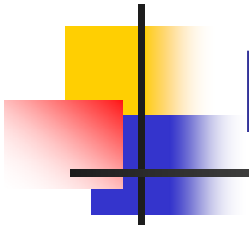
Filters

- A filter is a boolean expression evaluated for each value in the given *Extent* or *Collection*.
- Filters use syntax for boolean expressions that is familiar from Java.
- A filter can access the fields of a class even if they are `private`, but cannot invoke methods.
- A filter is defined for a query either using newQuery() or:
  - `void setFilter(String filter);`



Operators

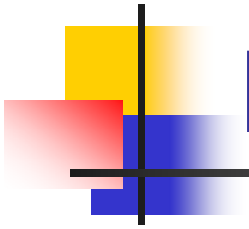
- Equality:
  - ==
  - !-
- Comparison:
  - <
  - <=
  - >
  - >=
- Boolean:
  - &
  - &&
  - |
  - ||
  - !
- In JDOQL, as there are no side-effects:
  - & ≡ && (conditional and)
  - | ≡ || (conditional or)



Examples - 1

- Find all trains that start at `argv[0]` and end at `argv[1]`:

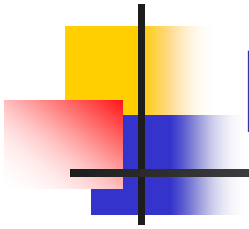
```
Query q = pm.newQuery(Train.class);
q.declareParameters("String src, String dst");

q.setFilter("source.name == src && dest.name == dst");

Collection result = (Collection)
    q.execute(argv[0], argv[1]);
```



Examples - 2

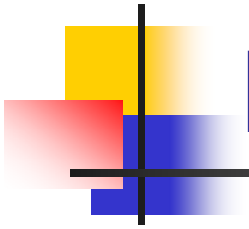
- Find all the trains that visit argv[0]:

```
System.out.println("Visit " + argv[0]);
Query q = pm.newQuery(Train.class);
q.declareParameters("String p");

q.declareVariables("Visit v");

q.setFilter("(route.contains(v) && v.place.name == p)");
result = (Collection) q.execute(argv[0]);
```



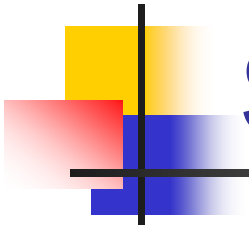
Navigation in JDOQL

- Navigating through collections:
  - `boolean contains(Object o)` is used with an *and* expression to yield true if at least one collection element matches the condition.
  - The parameter to `contains()` is a variable that has to have been declared using `declareVariables()` on *Query*.
- Path expressions navigate through single-valued relationships, as in Java.



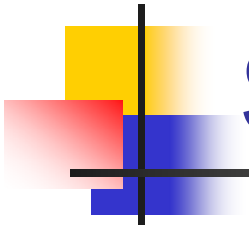
Not in JDOQL

- JDOQL in JDO 1.0 does not support:
  - Joins.
  - Aggregation.
  - Nested queries.
  - Updates.
  - Interactive queries.
  - Multiple programming-language embedding.



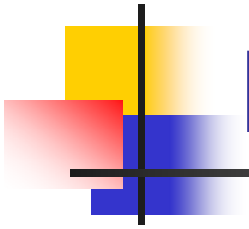
Summary: JDO

- JDO provides a standard for direct storage for Java objects, using object or relational stores.
- Java persistence is popular; there are other approaches (e.g. http://www.hibernate.org/).
- JDO is pragmatic, with straightforward support for (almost) orthogonal persistence, in quite a compact specification.
- We have had good experiences of JDO in complex applications (http://img.cs.man.ac.uk/gims).



Summary: Object Databases

- The relational model with JDBC does not provide close integration of the data model with the Java type system.
- Many applications can be modelled more naturally using objects than relational databases.
- Object databases have replaced relational solutions only rarely: they are used in niche markets, for data-intensive application development.



Further Reading

- D. Jordan, C. Jones, Java Data Objects, O'Reilly, 2003.
- D. Jordan, JDOQL: The JDO Query Language:
  - http://www.objectidentity.com/images/jdoql.pdf