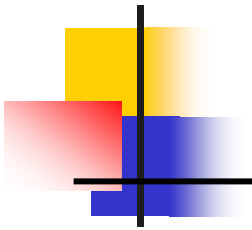
An Introduction to Java Data Objects

David Jordan

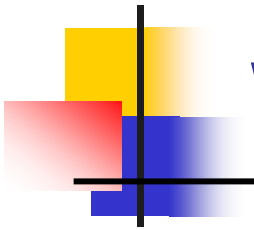
Object Identity, Inc.

www.objectidentity.com



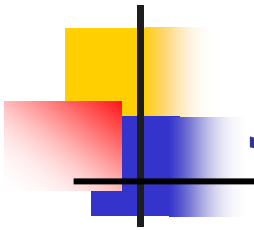
Presentation Overview

- JDO Standard
- Defining your model
- Class enhancement
- Establish datastore connection and transaction context
- Access/manipulate persistent objects
- Access *via* extents and queries
- Compare JDO to alternatives



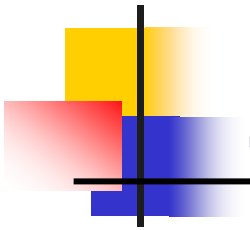
What Is JDO?

- Transparent persistence of Java object models in transactional datastores
- Became a standard in March 2002 *via* JCP
- Uses Java data model and language
  - Your Java classes serve as data model
  - Navigate your data using references & collections
  - Query using your Java model, Java operators
- Objects are auto-mapped to/from database
- Object cache managed automatically



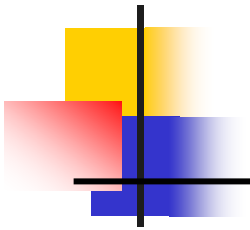
JDO Resources

- On the Web
  - http://java.sun.com/products/jdo
  - http://access1.sun.com/jdo
  - http://www.jdocentral.com
- In Print
  - Four JDO books now available



Just a Few of the JDO Vendors

- Relational database
  - SolarMetric, Libelis, Hemisphere Tech, Object Industries, Object Frontier, Exadel, SAP, Signsoft
- Object database
  - FastObjects, ObjectDB, Versant, Progress Software
- Open source
  - JBossDO, Jarkarta OJB, Speedo, TJDO, XORM

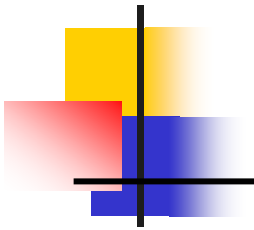


Small Number of Interfaces

- JDO API defined in package javax.jdo
- Interfaces
  - `PersistenceManagerFactory`
  - `PersistenceManager`
  - `Transaction`
  - `Extent`
  - `Query`
  - `InstanceCallbacks`
- Class: `JDOHelper`

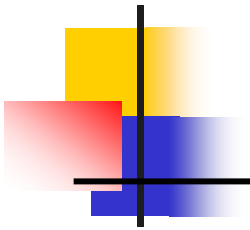
JDO Exception Hierarchy

- java.lang.RuntimeException
  - JDOException
    - JDOCanRetryException
      - JDODataStoreException
      - JDOUserException
        - JDOUnsupportedOptionException
    - JDOFatalException
      - JDOFatalUserException
      - JDOFatalInternalException
      - JDOFatalDataStoreException
        - JDOOptimisticVerificationException



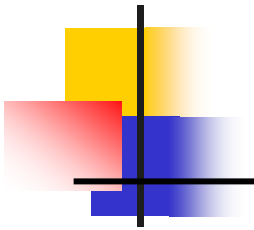
Defining Your JDO Model

- JDO model based on
  - Application's Java object model
  - XML metadata
- Don't use JDO-specific types
  - Specific standard Java types are supported
  - Persistent classes don't need to import any JDO
- Source not required to persist a class
- Classes to persist must be enhanced
- Persistent classes must have null constructor



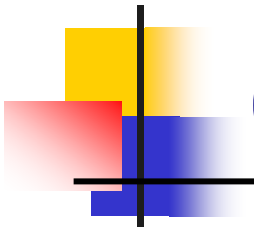
Class and Field Modifiers

- All Java class and field modifiers supported
  - `public, private, protected`
  - `abstract, synchronized, volatile`
  - `static, final, transient`
- Fields never stored in datastore
  - `final` (values never change)
  - `static` (only one instance per class)
- Fields not stored by default
  - `transient` (can override in XML metadata)



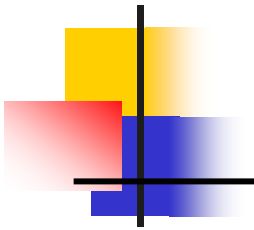
Field Types

- Primitives (`boolean`, `byte`, `int`, `float` …)
- Wrappers (`Boolean`, `Byte`, `Integer` …)
- java.lang: `String`, `Number`, `Object`
- java.util: `Date`, `Locale`
- java.math: `BigInteger`, `BigDecimal`
- Persistent class references
- Interface references



Collections

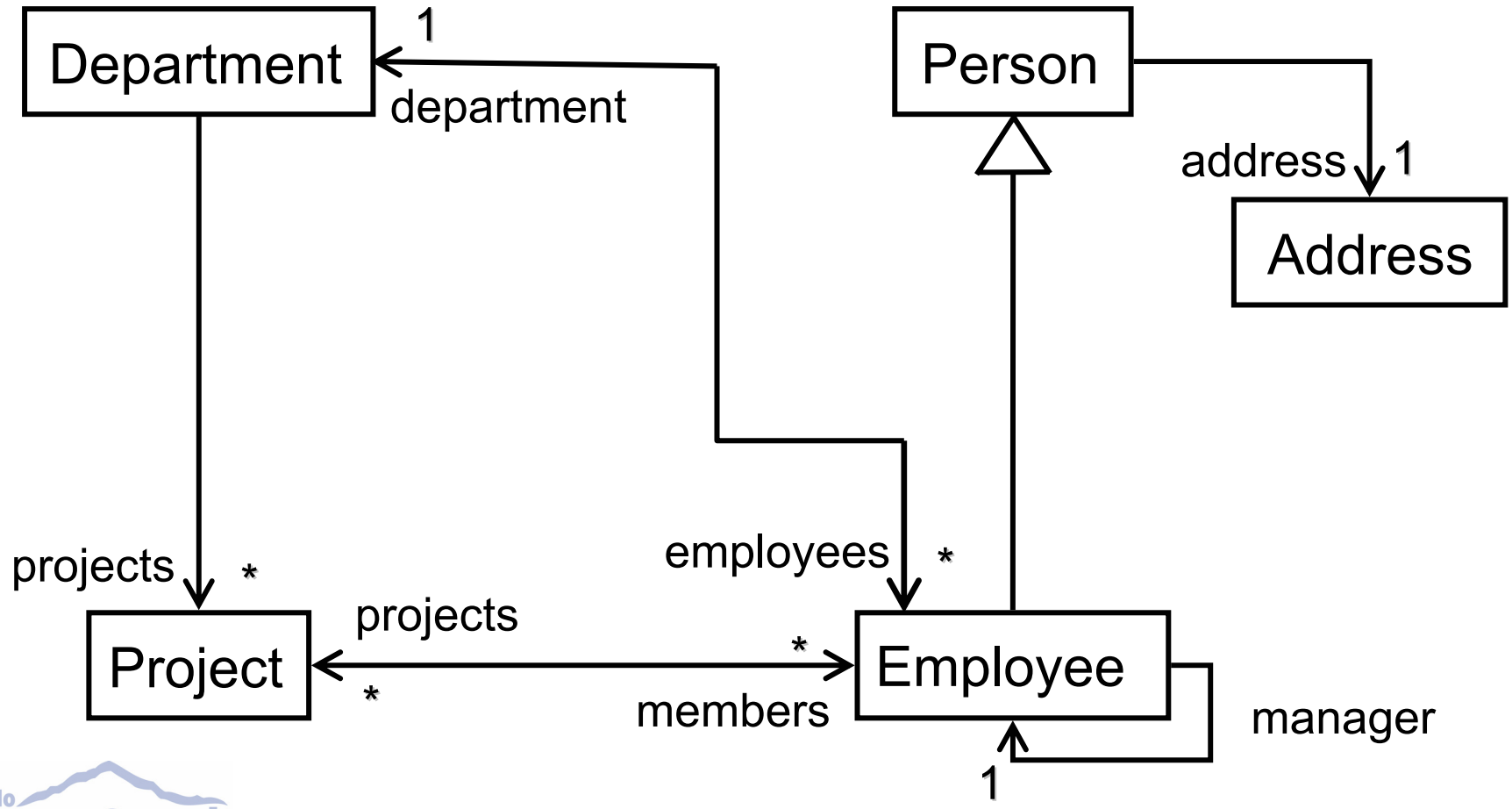
- Required
  - `Collection`, `Set`, `HashSet`
- Optional
  - `arrays`, `Hashtable`, `Vector`
  - `TreeSet`
  - `List`, `LinkedList`, `ArrayList`
  - `Map`, `HashMap`, `TreeMap`
- Most of the optional collections are supported by most of the vendors

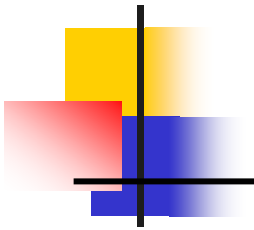


Inheritance

- Class inheritance is supported
- Persistence of a class is independent of its location in a class hierarchy
  - Persistent class can extend a transient class
  - Transient class can extend a persistent class
- Polymorphic references are supported
- Must declare class' nearest persistent base class in the class hierarchy in the metadata

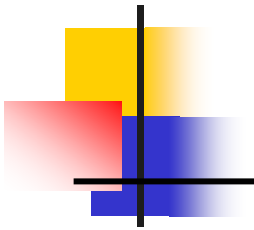
Example Schema: Company Database





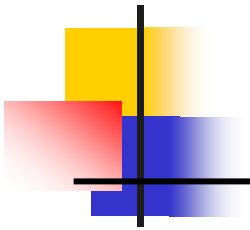
Person class

```
package company;
import java.util.Date;

public class Person {
  private String   firstname;
  private String   lastname;
  private Address  address;
  private Date     birthdate;

  protected Person() { }  // must have null constructor
  // other methods including get/set methods
}
```



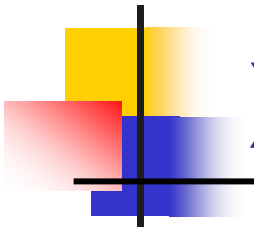
Employee class

```
package company;
import java.util.*;

public class Employee extends Person {
  private long       empid;
  private Date       hiredate;
  private float      weeklyhours;
  private Department department;
  private Employee   manager;
  private HashSet    projects; //element: Project

  // constructors and methods
}
```



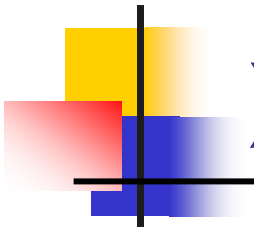
Department class

```
package company;
import java.util.*;

public class Department {
  private int     deptid;
  private String  deptname;
  private HashSet employees; // element:Employee
  private HashSet projects;  // element:Project

  public Department() { }

  // other constructors and methods
}
```



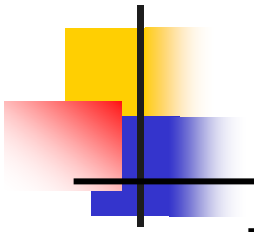
XML Metadata

- JDO metadata is specified in XML file
- Metadata needed to specify
  - Which classes are persistent
  - Persistence info not expressible in Java
  - Override default persistence behaviors
  - Enable vendor-specific features
- Can be
  - Specified with a GUI or by hand
  - Generated by a tool



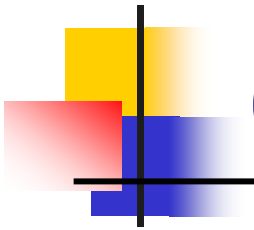
XML Metadata for Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="company">
    <class name="Address" />
    <class name="Person" />
    <class name="Employee"
      persistence-capable-superclass="company.Person">
      <field name="projects">
        <collection element-type="company.Project"/>
      </field>
    </class>
```



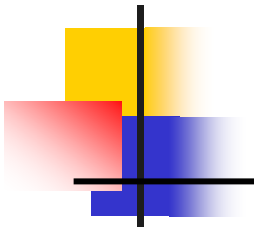
Metadata (Continued)

```
    <class name="Department">
      <field name="employees">
        <collection element-type="company.Employee" />
      </field>
       <field name="projects">
        <collection element-type="company.Project" />
      </field>
    </class>
    <class name="Project">
      <field name="members" >
        <collection element-type="company.Employee"/>
      </field>
    </class>
</package>
</jdo>
```



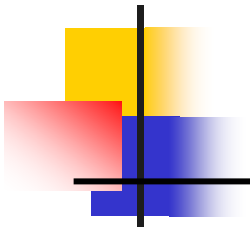
Object / Relational Mappings

- Approaches
  - Java model: generate relational schema
  - Relational schema: generate object model
  - Define mapping between existing object model and relational schema
- OR mappings are vendor-specific in JDO 1.0
- OR mappings typically placed in metadata
- Java source not impacted by OR mappings
- JDO 2.0 to standardize OR mappings



Example OR Mapping

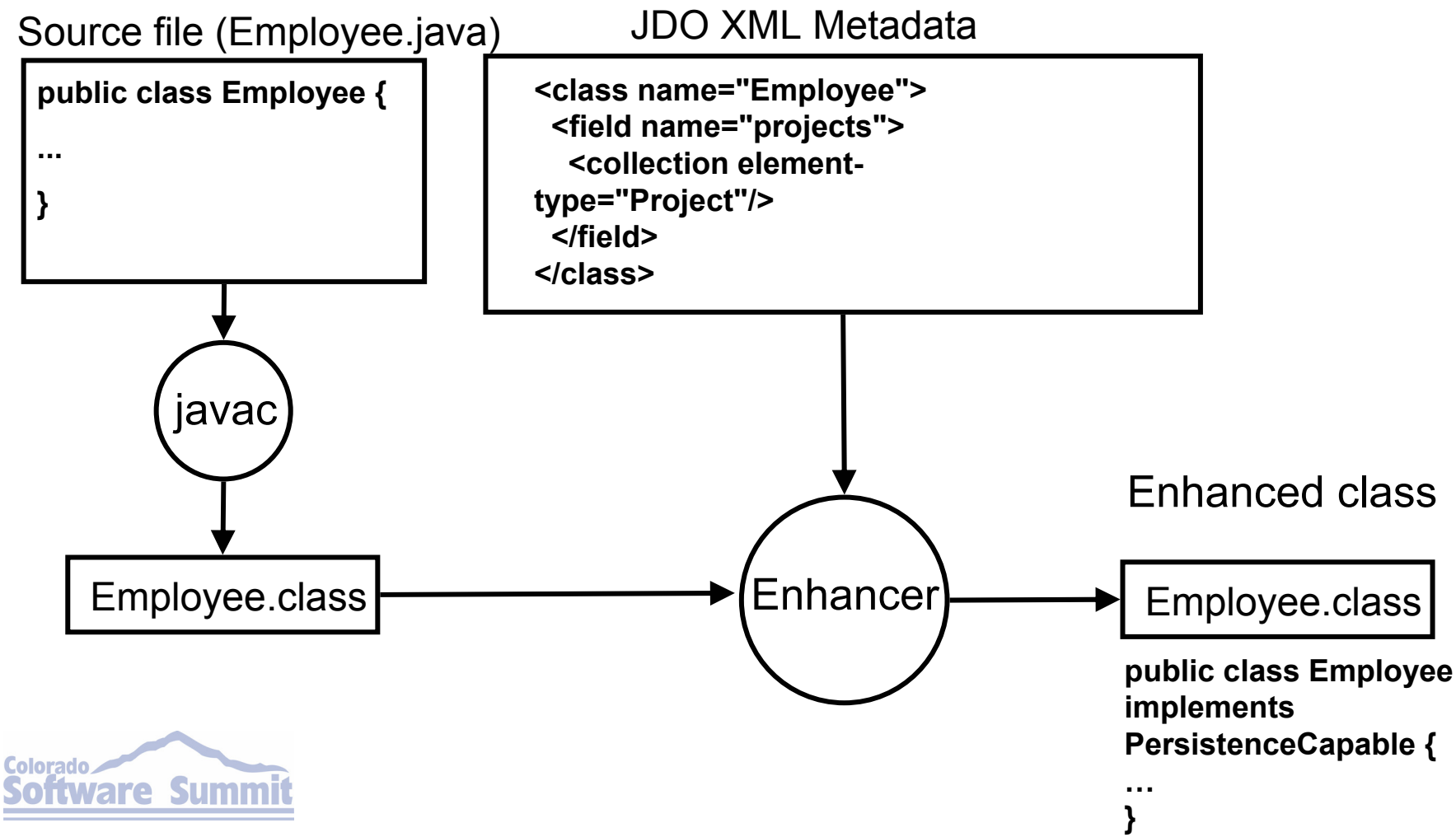
```
<class name="Employee"
    persistence-capable-superclass="company.Person">

    <extension key="table" value="EmpTable"
               vendor-name="X" />

    <field name="manager">
       <extension key="column" value="mgr"
                  vendor-name="X" />
    </field>
</class>
```

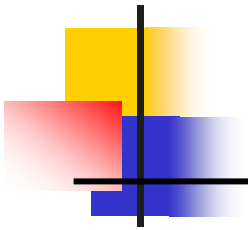


Class Enhancement

- Data/methods added to persistent classes to support database transparency
- Several approaches to enhancement
  - Hand-code necessary code enhancements
  - Code generated by tool at source level
  - Use enhancer, code added at class file level
- Interface standardized between PersistenceManager and persistent instances
- Binary compatibility across JDO enhancers

Enhancer

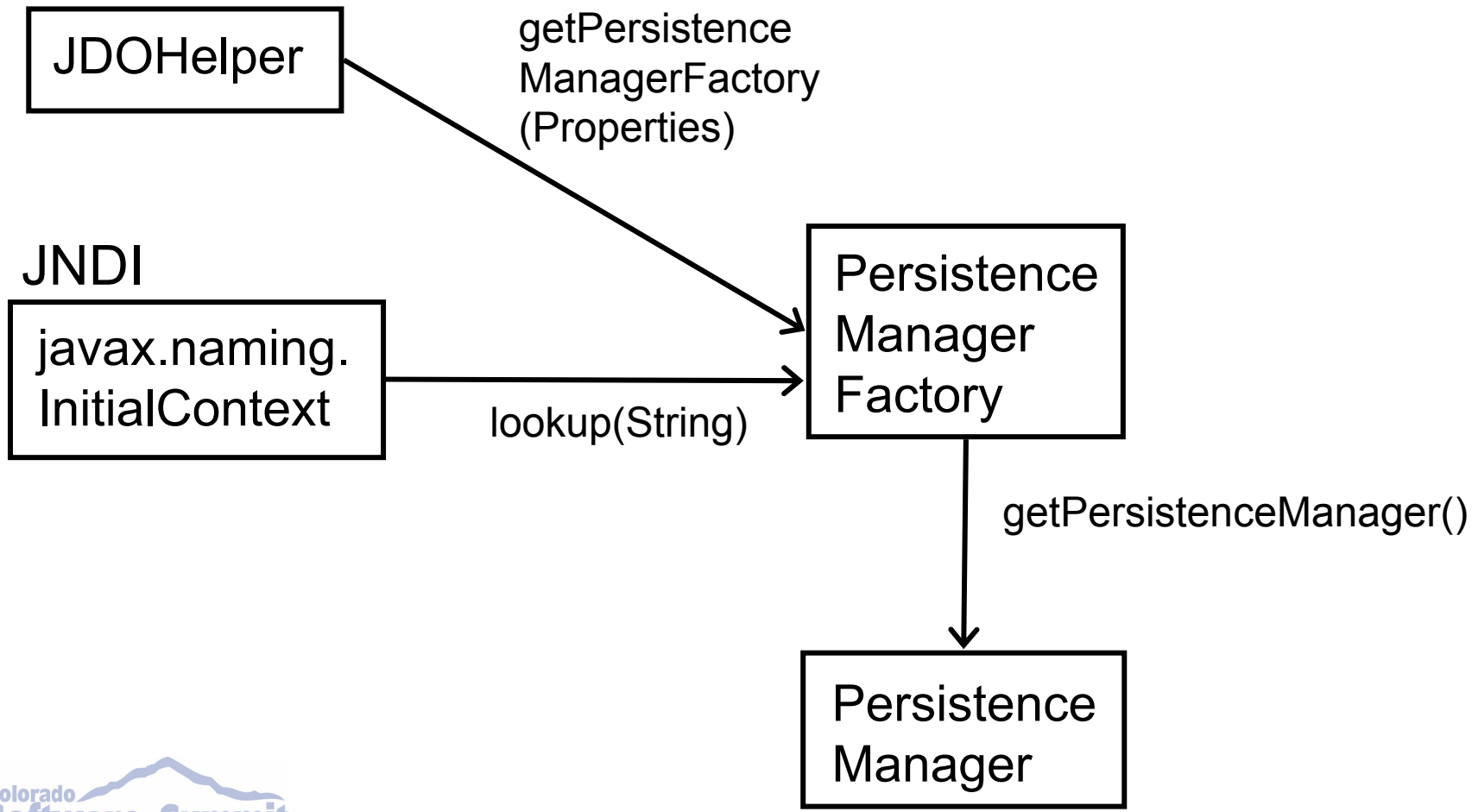


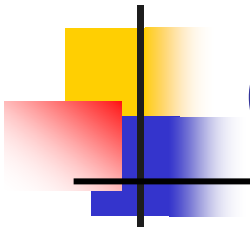


PersistenceManager (PM)

- Primary JDO interface for managing objects
- Provides methods to
  - Make instances persistent, `makePersistent()`
  - Delete instances, `deletePersistent()`
- Manages a cache of objects in transaction
  - Manages their identity and lifecycle
  - One copy of a persistent instance in the cache
- Manages single transaction context
- Used to create `Query` instances

Acquire a PersistenceManager





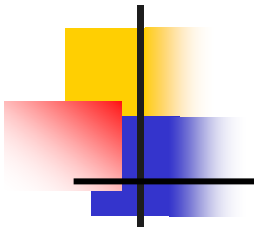
Get PMF *via* Properties

Property file:

```
javax.jdo.PersistenceManagerFactoryClass=
      com.sun.jdori.fostore.FOStorePMF
javax.jdo.option.ConnectionURL=fostore:database/fostore
javax.jdo.option.ConnectionUserName=dave
javax.jdo.option.ConnectionPassword=jdo4me
```

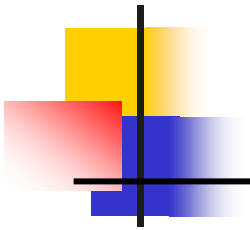
Necessary code:

```
PersistenceManagerFactory pmf = null;
InputStream propStream =
              new FileInputStream("jdo.properties");
Properties props = new Properties();
props.load(propStream);
pmf = JDOHelper.getPersistenceManagerFactory(props);
```



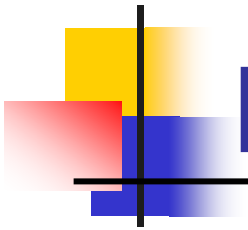
Establish Transaction Context

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

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

// put database transaction logic here

tx.commit(); // or rollback() to abort transaction

// possibly multiple transactions
pm.close();
```



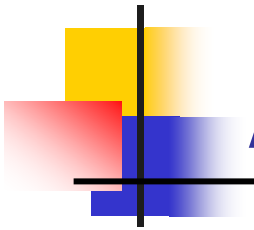
Persistence of Instances

- Explicit persistence
  - `PersistenceManager` methods:
    - `void makePersistent(Object o);`
    - `void makePersistentAll(Object[] a);`
    - `void makePersistentAll(Collection c);`
- Persistence-by-reachability
  - Transient instance of a persistent class referenced by a persistent instance becomes persistent at transaction commit
  - Can persist an entire object graph



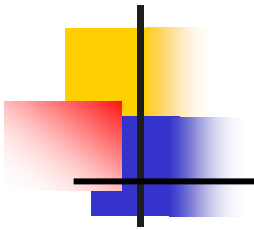
Example of Persistence

```
Transaction tx = pm.currentTransaction();
tx.begin();
Department dev = new Department(1000, "Development");
pm.makePersistent(dev);

Address addr =
  new Address("100 Elm Street", "Cary", "NC", "27513");
Employee emp = new Employee("John", "Doe", addr);
dev.addEmployee(emp);

addr = new Address("250 Walnut Street",
                   "Apex", "NC", 27514);
emp = new Employee("Bob", "Jones", addr);
dev.addEmployee(emp);
tx.commit();
```



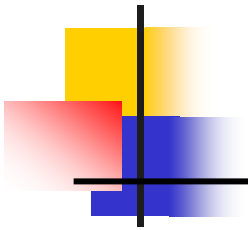
Accessing Instances

- Iterate an extent or issue a query to access some initial objects
- Application can access related instances
  - Simply navigate through a reference
  - Iterate on a collection of references
- JDO runtime loads instances on demand
  - One copy of each instance per transaction
- You can directly access and modify fields



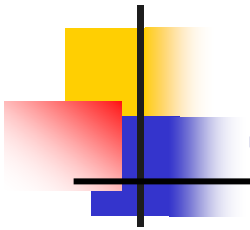
Modifying Instances

- You can directly modify a field
- Modify references and collections to change the relationships among objects
- Instances modified are automatically marked as updated
- All modifications to instances and their interrelationships are automatically propagated to the database at commit



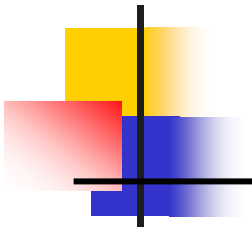
Using an Extent

```
Extent depts = pm.getExtent(Department.class, true);
Iterator iter = depts.iterator();
while( iter.hasNext() ){
    Department d = (Department) iter.next();
    System.out.print("Department " + d.getName() );
    Set employees = d.getEmployees();
    Iterator empIter = employees.iterator();
    while( empIter.hasNext() ){
        Employee e = (Employee) empIter.next();
        System.out.print("\t");
        System.out.print(e.getFirstName());
        System.out.println(e.getLastName());
    }
}
depts.close(iter);
```



JDO Query Language (JDOQL)

- Use Java object model identifiers in filter
- Use Java expressions and operators
- Apply a Boolean filter to a set of candidates
  - `Extent`
  - `Collection`
- Candidates are of a particular class
  - Establishes a scope for names in the query
  - Use import statements to include other types
- Instances that evaluate true are in the result



Parameters, Navigation, Order

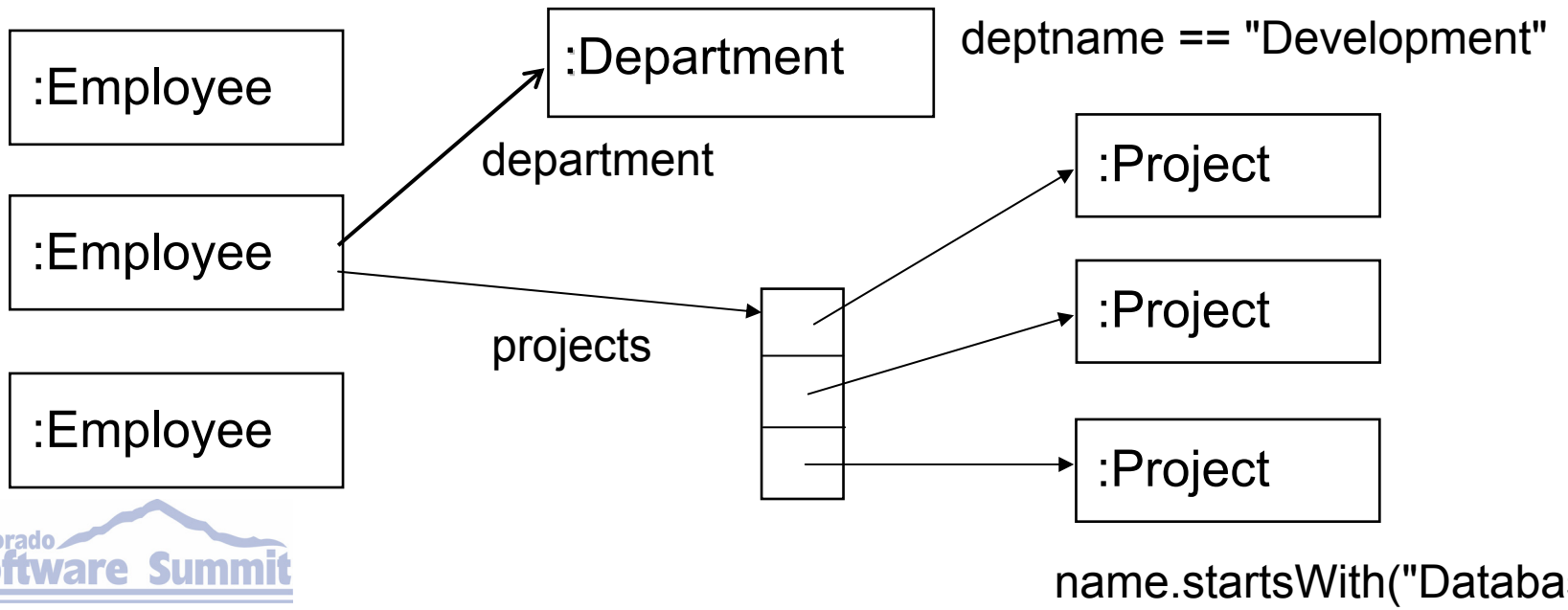
- Query instance created with one of several `PersistenceManager.newQuery()`
- Parameters used to provide run-time values
- Can navigate through references/collections
  - Variables used to reference collection elements
- Result can be ordered
  - Similar to SQL order by

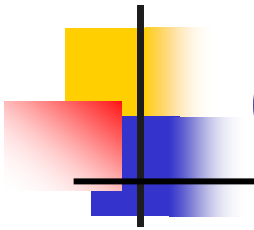
Operators in Query Filter

Oper	Descr	Oper	Descr	Oper	Descr
==	Equal	&	Boolean logical AND	+	Addition
!=	Not equal	&&	Conditional AND	+	String concatenation
<	Less than	|	Boolean logical OR	-	Subtraction, sign inversion
<=	Less than or equal	||	Conditional OR	*	Multiplication
>	Greater than	!	Logical complement	/	Division
>=	Greater than or equal	~	Unary bitwise compliment		

Example Query

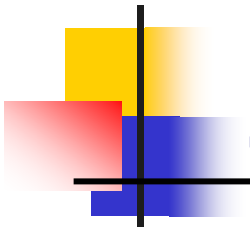
Retrieve all employees
in a department named "Development"
that work on a project whose
name starts with the word "Database"





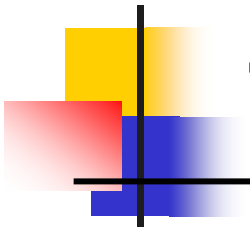
Code for the Query

```
String filter =
    "department.deptName == dname && " +
    "projects.contains(proj) && " +
    "proj.name.startsWith(pname)";
Extent emps = pm.getExtent(Employee.class, true);
Query query = pm.newQuery(emps, filter);
query.declareImports("import company.Project");
query.declareVariables("Project proj");
query.declareParameters("String dname, String pname");
query.setOrdering(
            "lastname ascending, firstname ascending");
Collection result = (Collection)
             query.execute("Development", "Database");
Iterator iter = result.iterator();
```



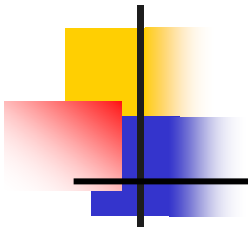
JDO and JDBC

- JDBC
  1. Transaction support
  2. Relational data model
  3. SQL queries, but with DBMS-specific dialects
  4. Targeted DBMSs: relational
- JDO
  1. Transaction support
  2. Java object model
  3. JDOQL language, standard language/syntax
  4. Targeted DBMSs: relational, object, others



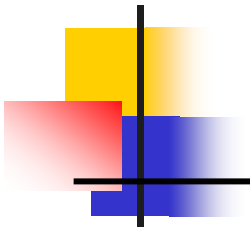
The Real Question to Ask…

- Do you want your application to manage information as objects or tables?
- Is SQL or Java more suited for your data?
- If you want to use objects and leverage the benefits of object oriented development …
  - You should seriously consider JDO
  - You can still use JDBC too
- If you want tables & SQL functionality…
  - Use JDBC



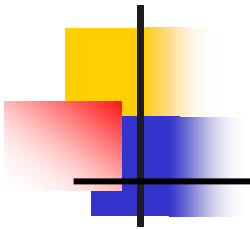
Alternatives to JDO

- Use a vendor's nonstandard, proprietary API
- Implement your own OR mapping
  - Development costs are prohibitive, ~30-40%
  - Resulting framework has limited capabilities
  - Requires high-level of staff expertise
- Use EJB Container Managed Persistence
  - EJB is much more complex than JDO
  - Lacks support for object features like inheritance
  - Entity Bean performance may be unacceptable



Benefits of JDO

- Developer productivity benefits
  - Provides mapping between objects & database
  - Developers stay focused on the object model
  - Application manages objects, not tables
  - Don't need database & model mapping experts
- Portability benefits
  - JDO provides binary compatibility across all JDO implementations and supported databases
  - JDO Query language consistent across impls.



Summary of JDO

- Very natural API for Java developers
  - Data model is Java
  - Access and navigation using Java constructs
  - Query language uses Java model and operators
- Lets developers to focus on the design of their Java object models
  - Without imposing modeling design constraints
- Small number of interfaces to learn