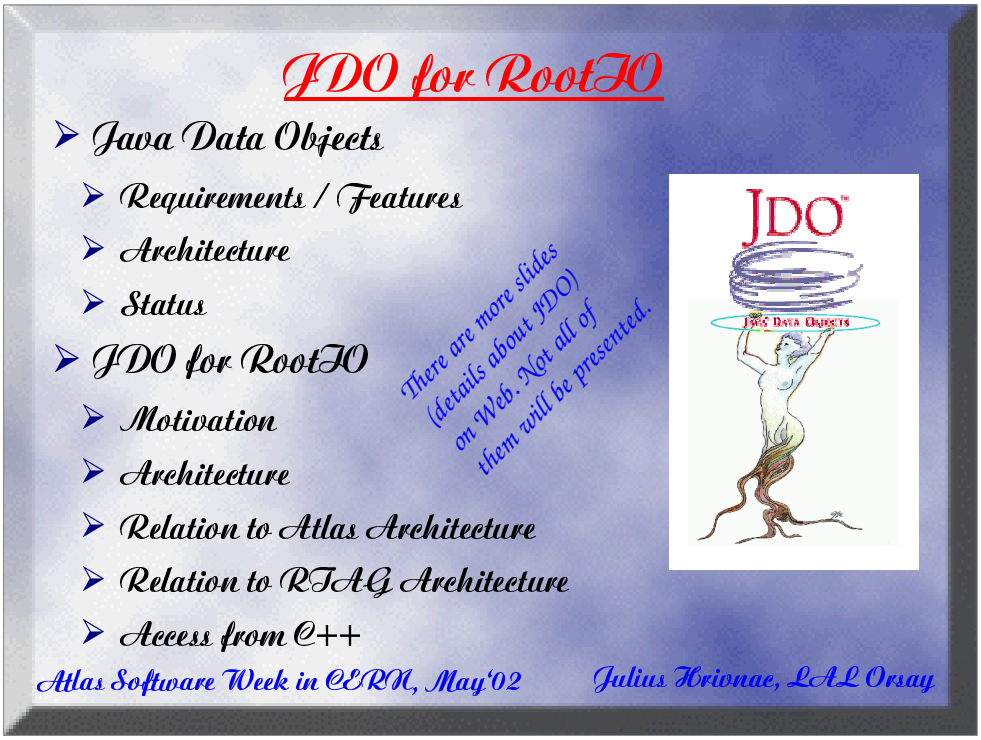
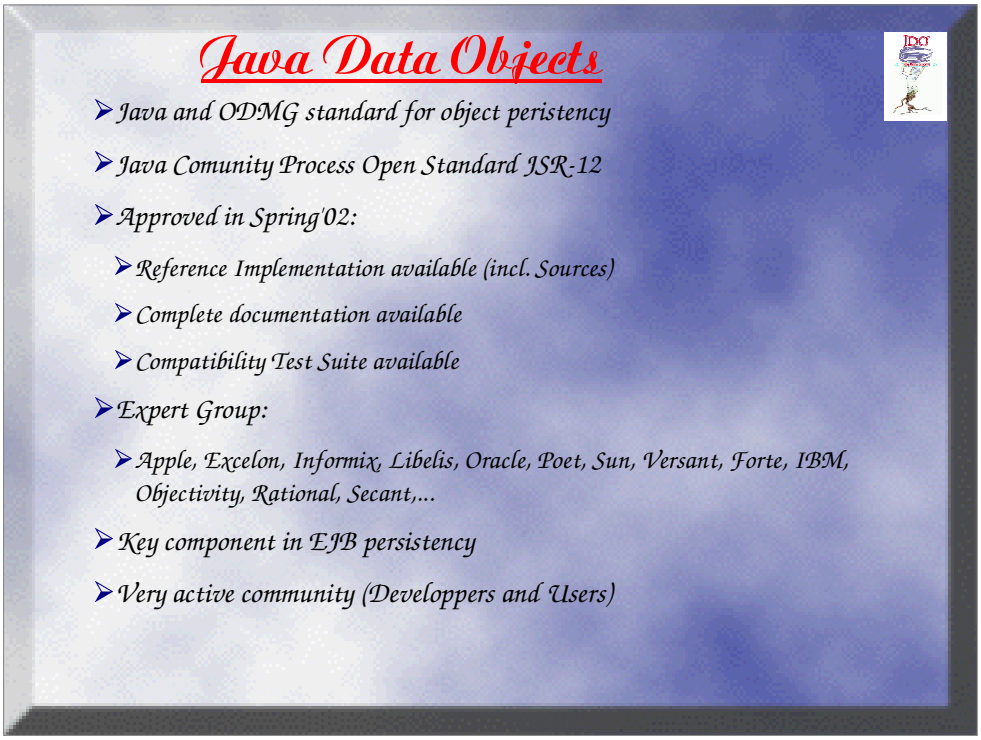
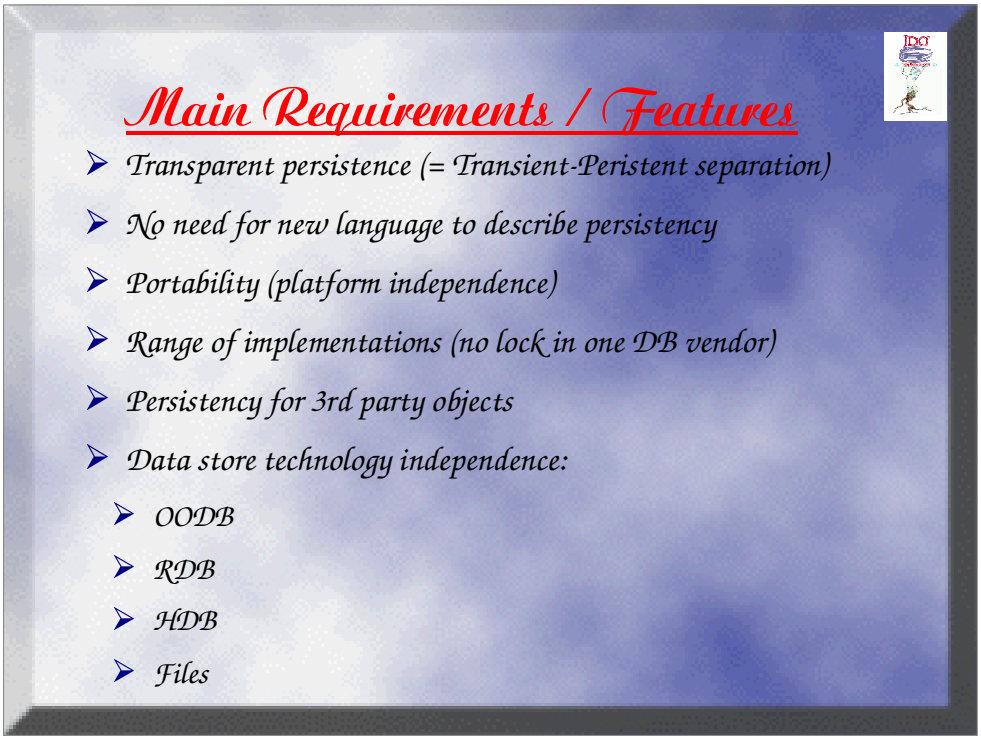


•Half yer ago (September'01): overview of JDO; today: concrete realistic proposal how to implement (Root) Persistency Service for both Java and C++ using JDO.
•In the Java Note (almost two years ago), we wrote that persistency is not resolved in Java. It is no more true. Java now offers persistency APIs superior to those of C++ (JDO, JDBS, jSQL, RootIO-Java, Objy-Java, Oracle-Java). Also Java 1.4 (Merlin) delivers significant improvement in Java IO speed.
•We are doing a lot of useless work, there are products which already satisfy our requirements.
•The slides describing basic JDO features are included (mostly copied from the presentation made to Atlas in September 2001) but mostly skipped during the presentation.



- Each Java standard should have Reference Implementation and Test Suite.
- Very active developpers community, many existing activities and products (both commercial and Free).

- Many commercial applications have free versions.
- New implementations / versions are comming very quickly.
- Castor is not 100% compatible.



- Transparent Persistence = Transien-Persistent Separation.

X

Transparent Persistence

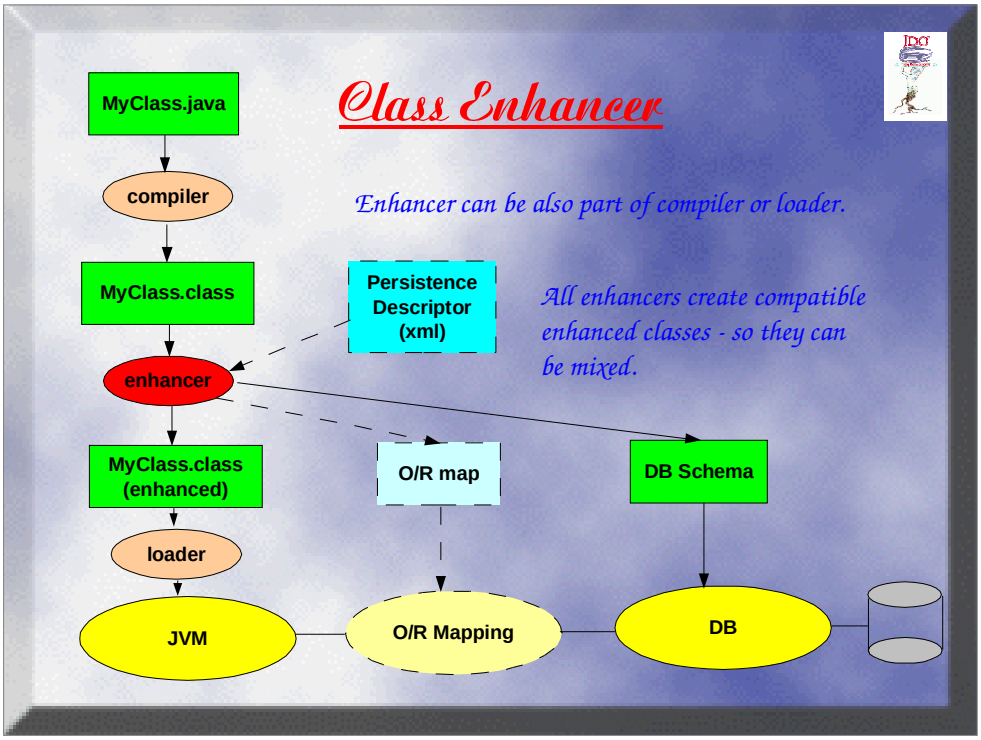
```
public class Hit {
  public Hit(double x,
             double y,
             double z) {
    m_x = x;
    m_y = y;
    m_z = z;
    }
  public double x() {
    return m_x;
    }
  public double y() {
    return m_y;
    }
  public double z() {
    return m_z;
    }
  private double m_x = 0;
  private double m_y = 0;
  private double m_z = 0;
  }
```

Transient class

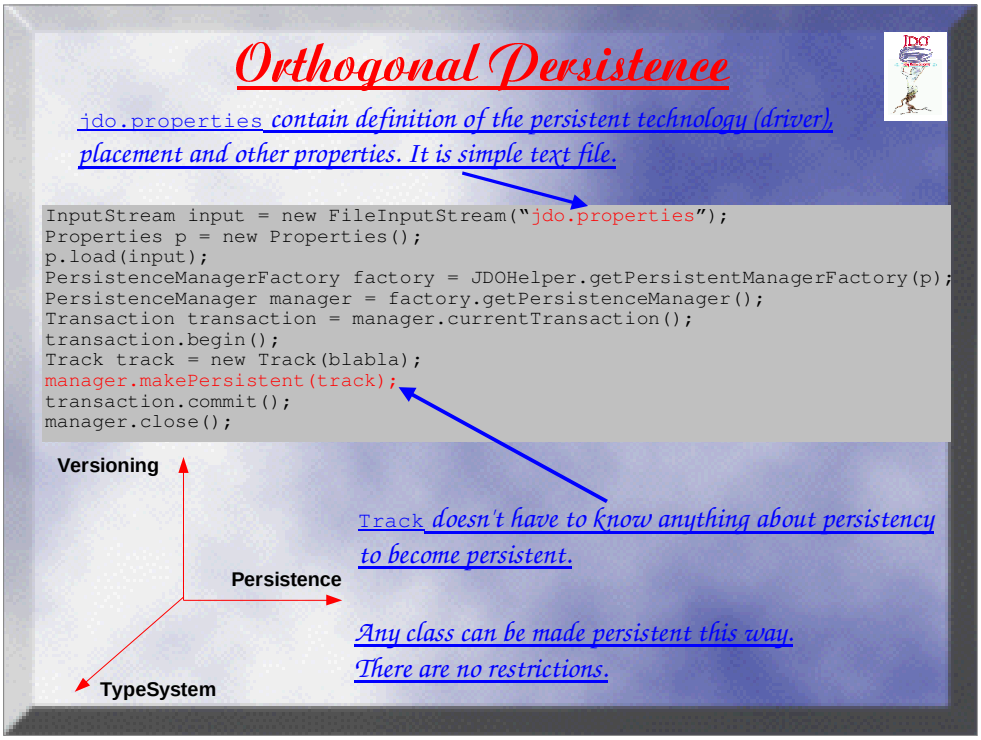
Persistent class

```
public class Hit {
  public Hit(double x,
             double y,
             double z) {
    m_x = x;
    m_y = y;
    m_z = z;
    }
  public double x() {
    return m_x;
    }
  public double y() {
    return m_y;
    }
  public double z() {
    return m_z;
    }
  private double m_x = 0;
  private double m_y = 0;
  private double m_z = 0;
  }
```

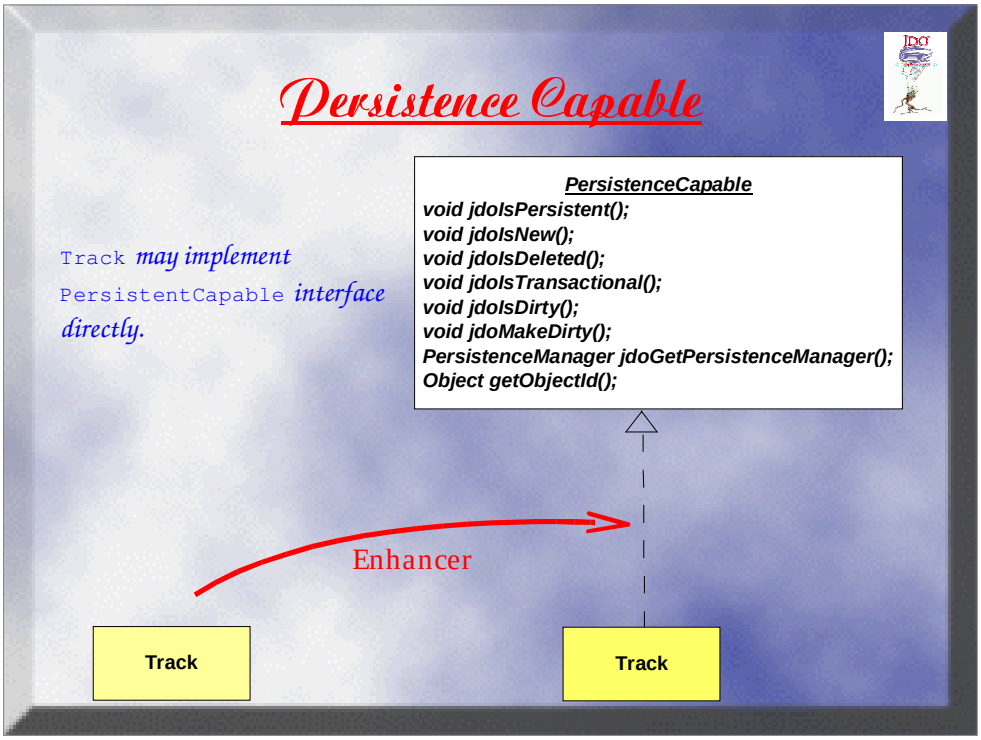
No additional code is required to make a class persistent.



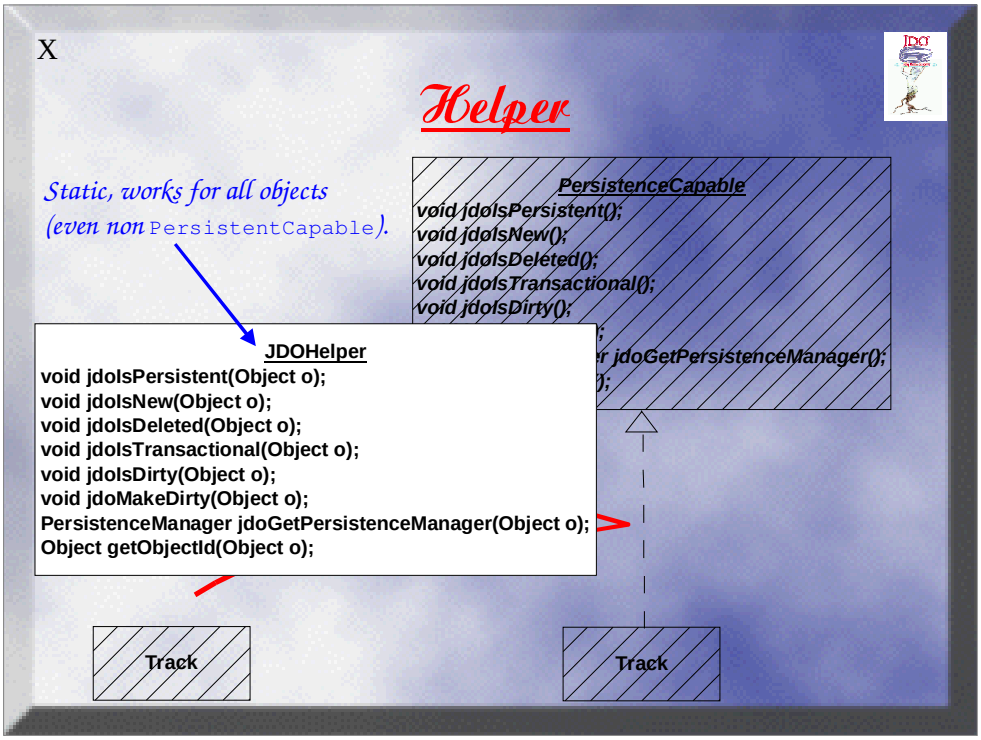
•This is the most easy (reference) implementation, but class can be enghanced at any stage:source postprocessing, while compiling, while loading,...
•All Enhancers should be compatible.
•Customisation: transient data, T-P mapping, clustering of data,...



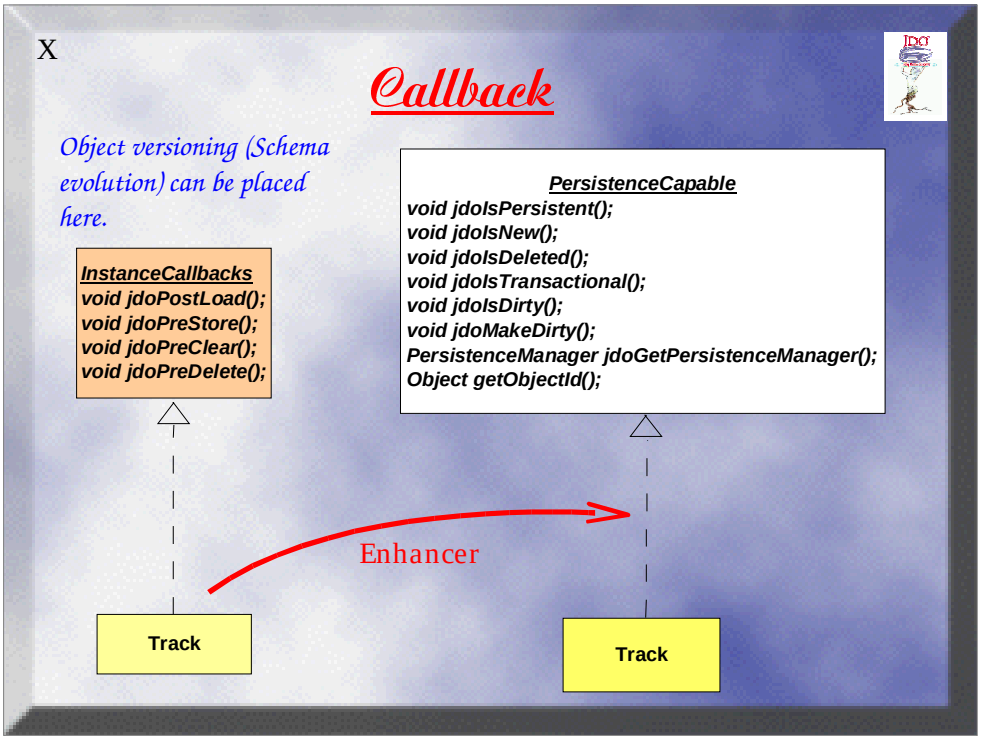
•Exactly the same source for all implementation of JDO, special properties defined in jdo.properties file 9and possibly in XML descriptions of classes).
•Class can be made persistent, if it is completely defined by its fields. So, for example, classes with native interface, Threads, Sockets or Files can't be made persistent.
•Strictly speaking, JDO doesn't implement exact Orthogonal Persistency, but the differentces are minor and irrelevent for us.
•Orthogonal versioning implements Class evolution (Scheme evolution), it uses similar architecture as Orthogonal persistence, but it is not much developed.
•Transaction management can be omitted.



•Enhancer changes normal class into PersistenceCapable class, but user can do it herself.
•Source (.java) doesn't know about persistency, run-time does know it.



- Helper is the prefered way of interaction with JDO.



•Used for extending persistency mechanism.
•Direct implementation can be usefull for example to generate common OID.

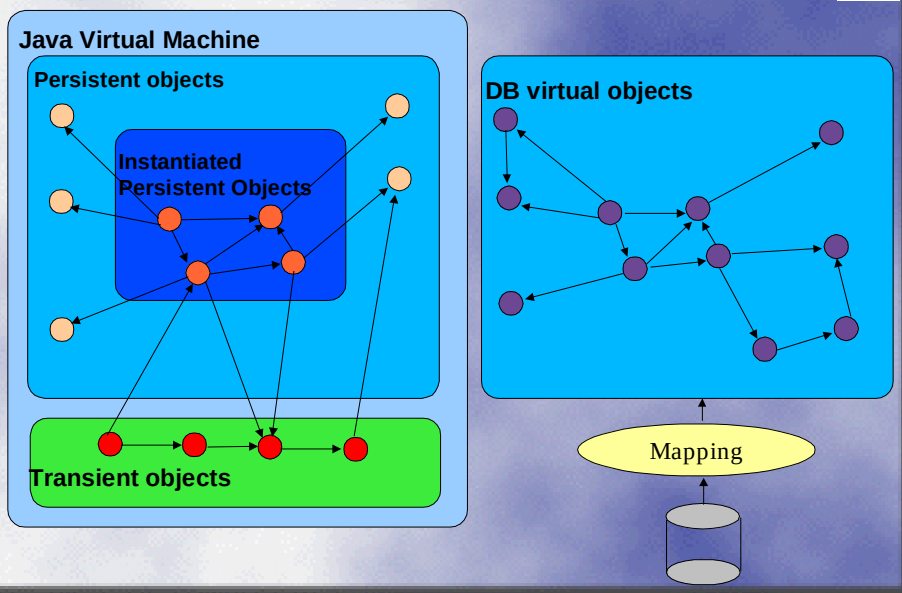
X

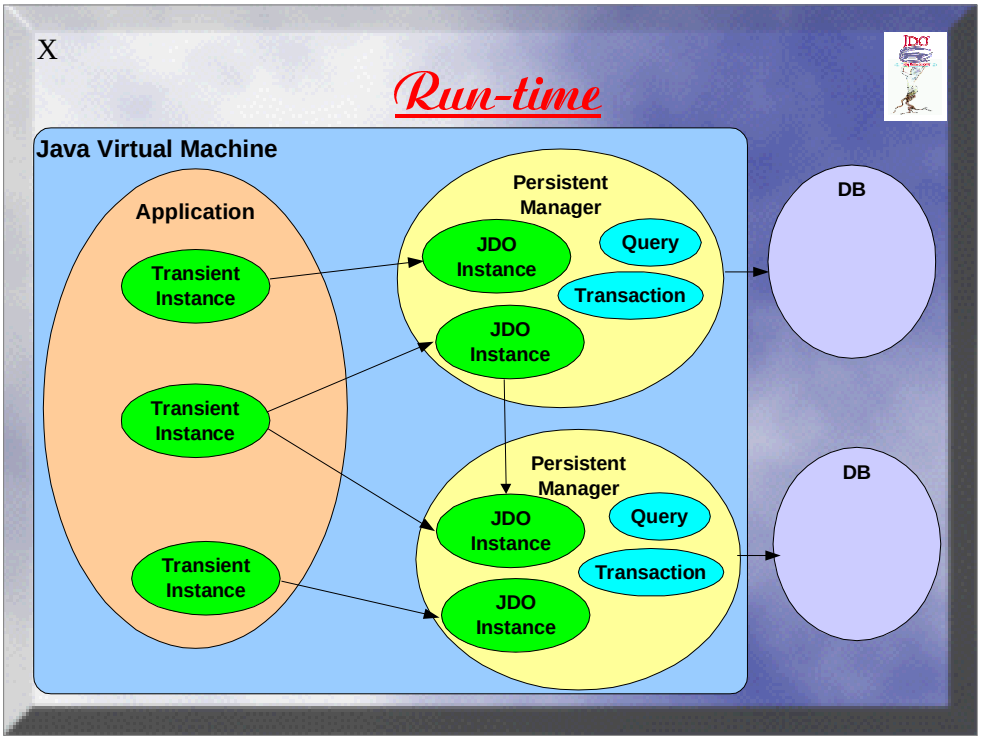
Objects

- *Classes can be:*
  - *Persistence-capable (enhanced, can have transient or persistent instances)*
  - *Transient (can't be persistent, system classes like Thread, classes with native components,...)*
  - *Persistence-aware (enhanced, can't be persistent, but can access public data of persistent classes)*
- *Objects can become persistent as:*
  - *First Class object (persistent by itself, it has its OID)*
  - *Second Class object (persistent due to its relationship to some First Class object, it doesn't have its OID, contained object)*
- *Fields can be:*
  - *Persistent (managed by JDO)*
  - *Transactional non-persistent (partly manged by JDO)*
  - *Non-transactional non-persistent (managed by application)*
  - *Grouped into fetch groups which are accessed together*

X

Object Model





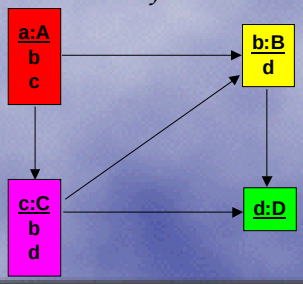
- Objects can be in different databases as all Enhancers should create compatible code.
- Database can be local or distributed (via application server).

X

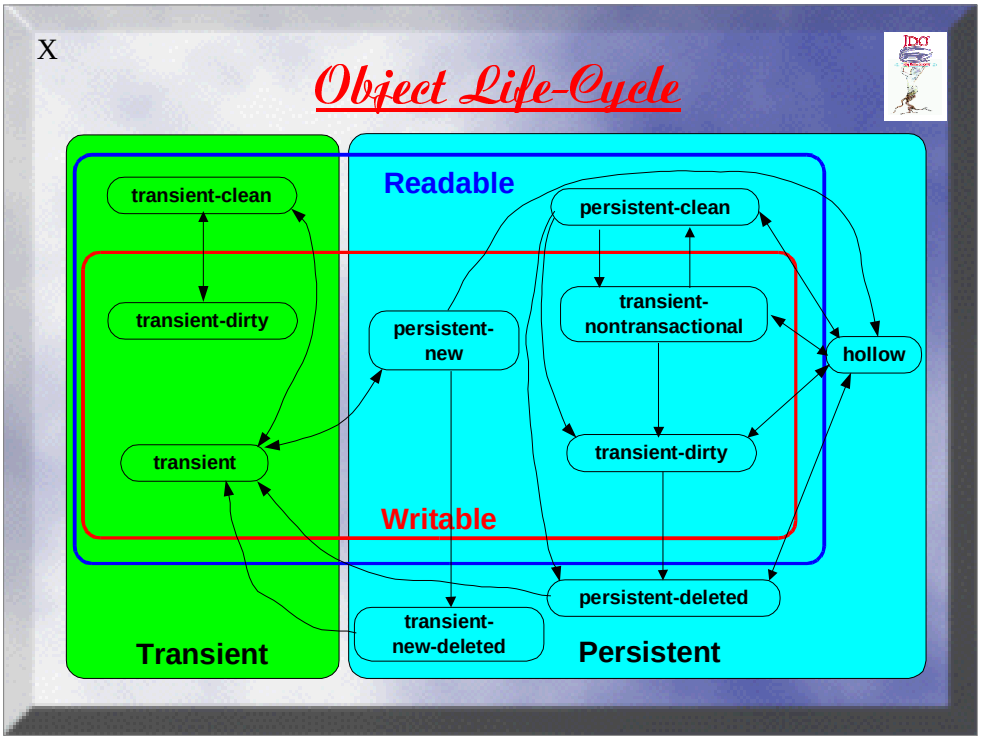
Persistence by Reachability

- *All objects referenced by persistent objects will also become persistent.*
- *Objects are loaded implicitely (lazily) when fields are accessed or by demand.*
  - *Fields can be pre-loaded/cached in groups (like in split Trees).*
- *Modified objects are implicitely updated in the database.*
- *Unreachable objects are removed from the database by a garbage collector.*
- *Embedded objects should be managed by their conteiner objects.*

```
A a = db.lookup(key);
B b = a.b();
C c = a.c();
D d = c.d();
```

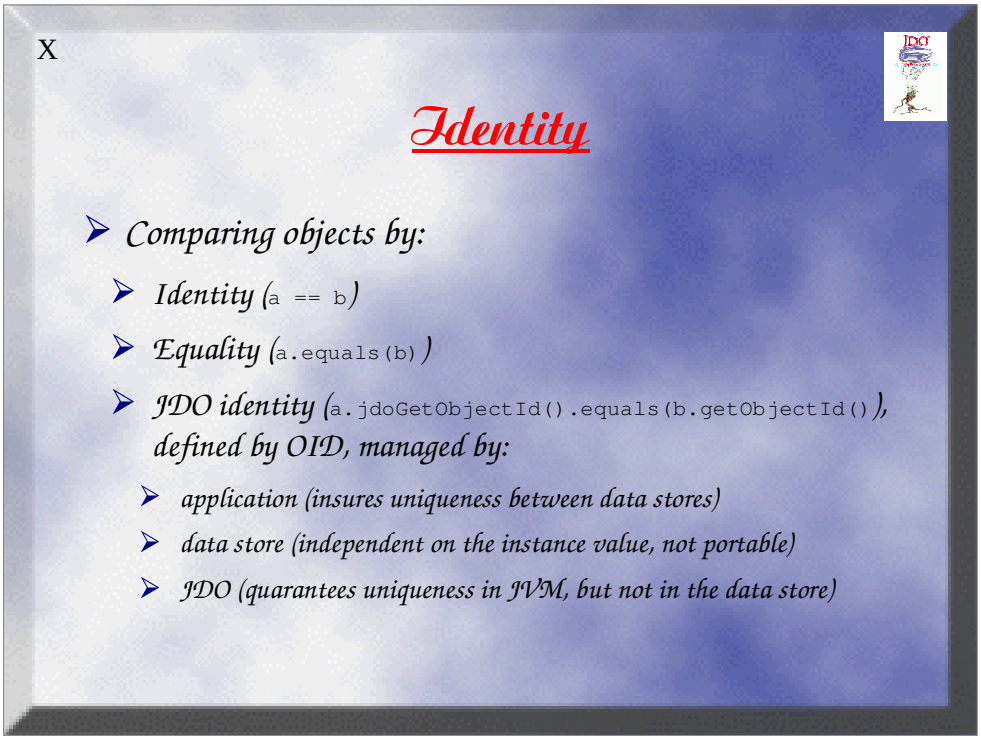


•Common concept for Java Databases.
•b should be First Class object, otherwise it can't be shared between a and c. Otherwise, one would have two copies of b. This is similar to StoreGate problems.



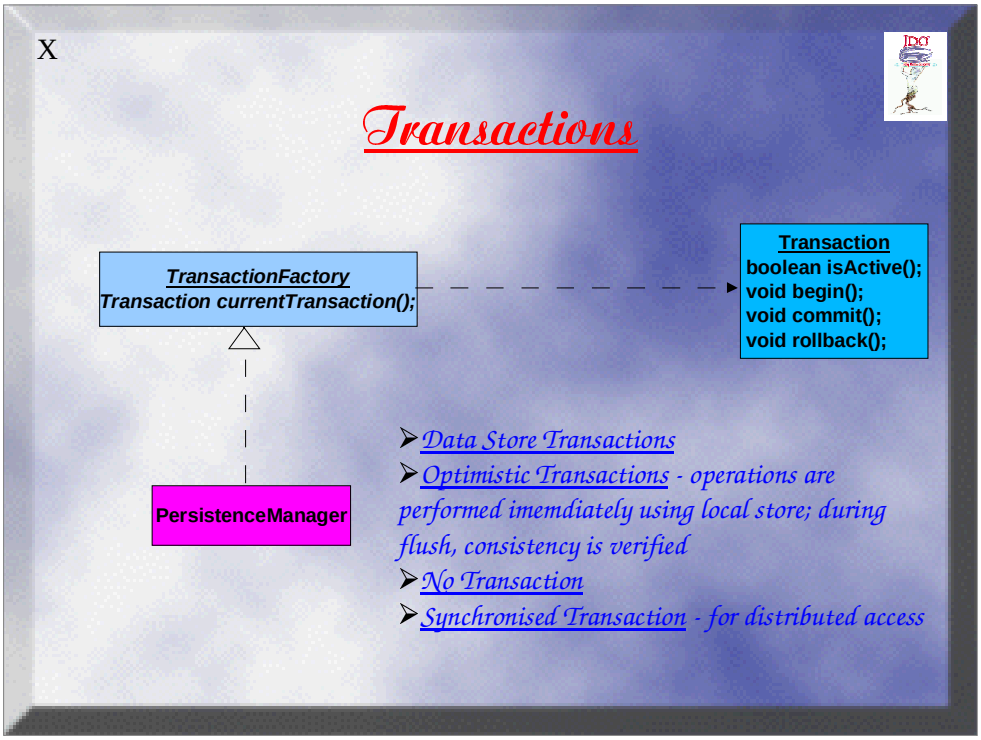
- It helps to understand and manage object' lifecycle.
- Transitions correspond to jdo methods.
- User doesn't care.

X



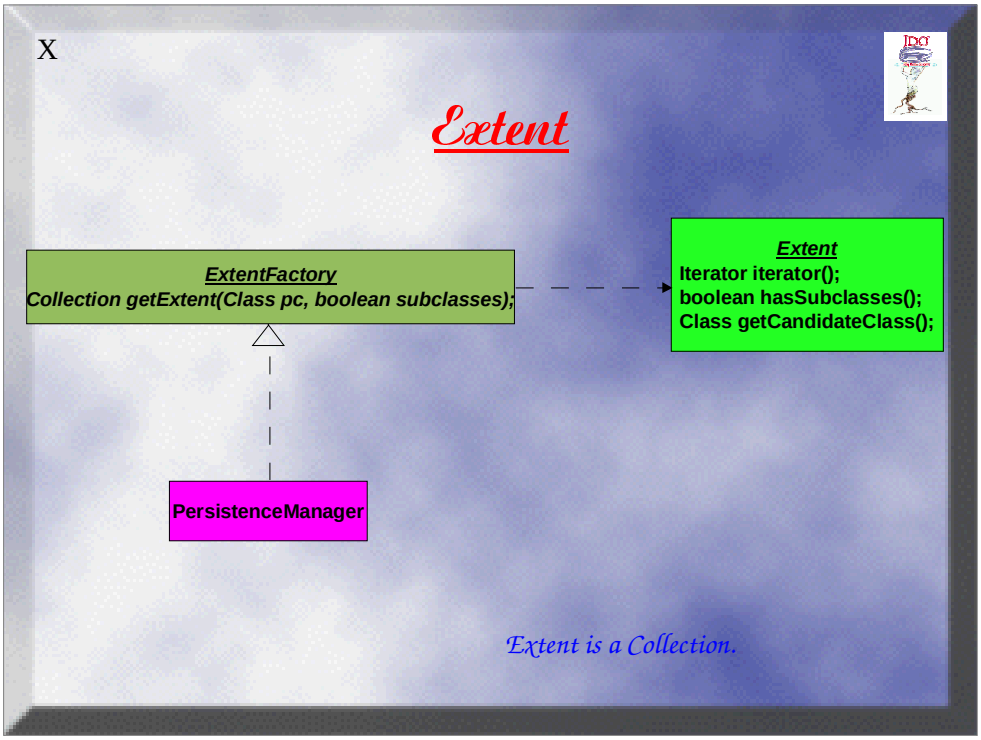
- Identity and Equality are standard Java concepts.
- To insure universal navigation, application identity should be used.

X

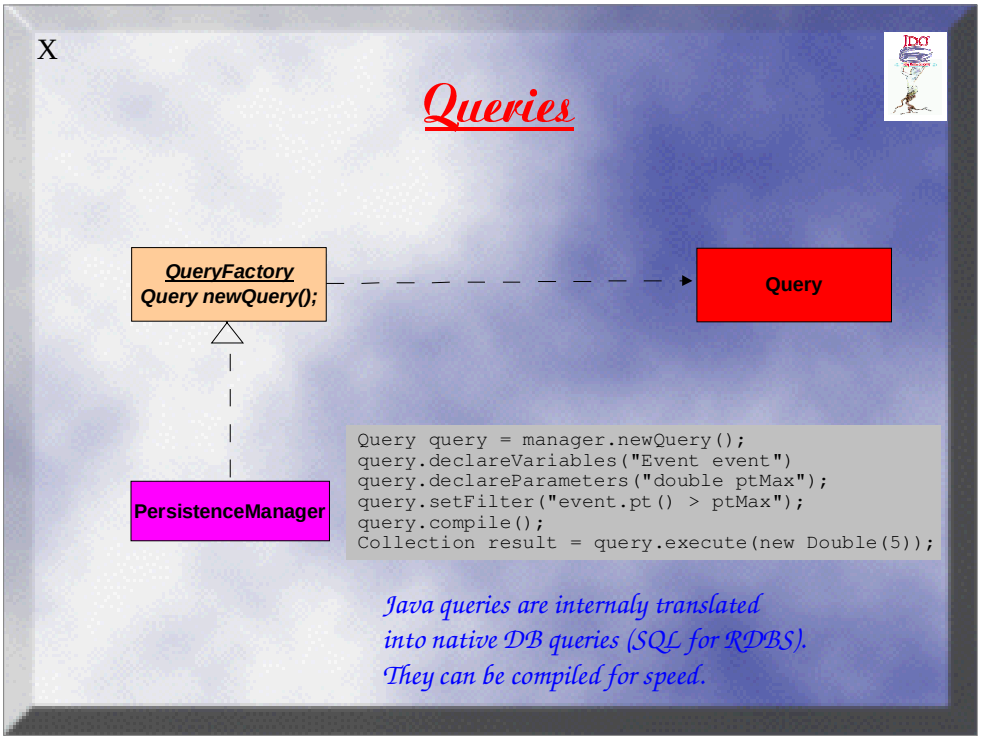


Transactions

- Data Store Transactions
- Optimistic Transactions - operations are performed imemdiately using local store; during flush, consistency is verified
- No Transaction
- Synchronised Transaction - for distributed access

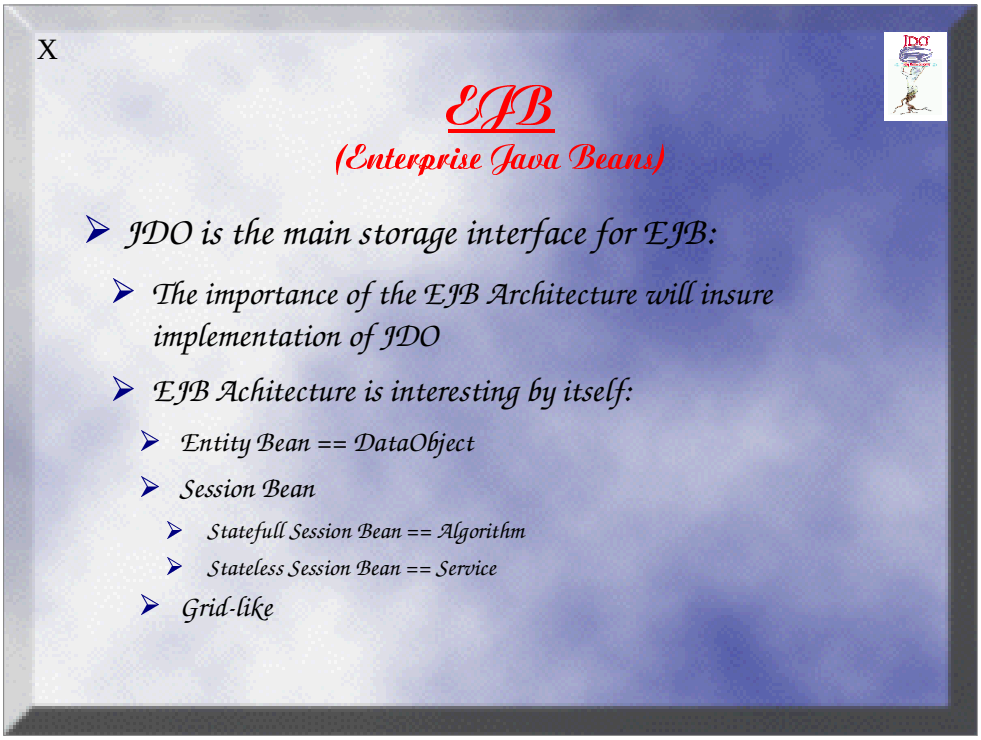


- Allows searching using object types.
- Similar concept exists in StoreGate.



- Queries are expressed in Java itself (not in additional language like SQL), but they have "set" semantics.
- Native DB queries for RDBS are efficient.
- Extent is used.

X



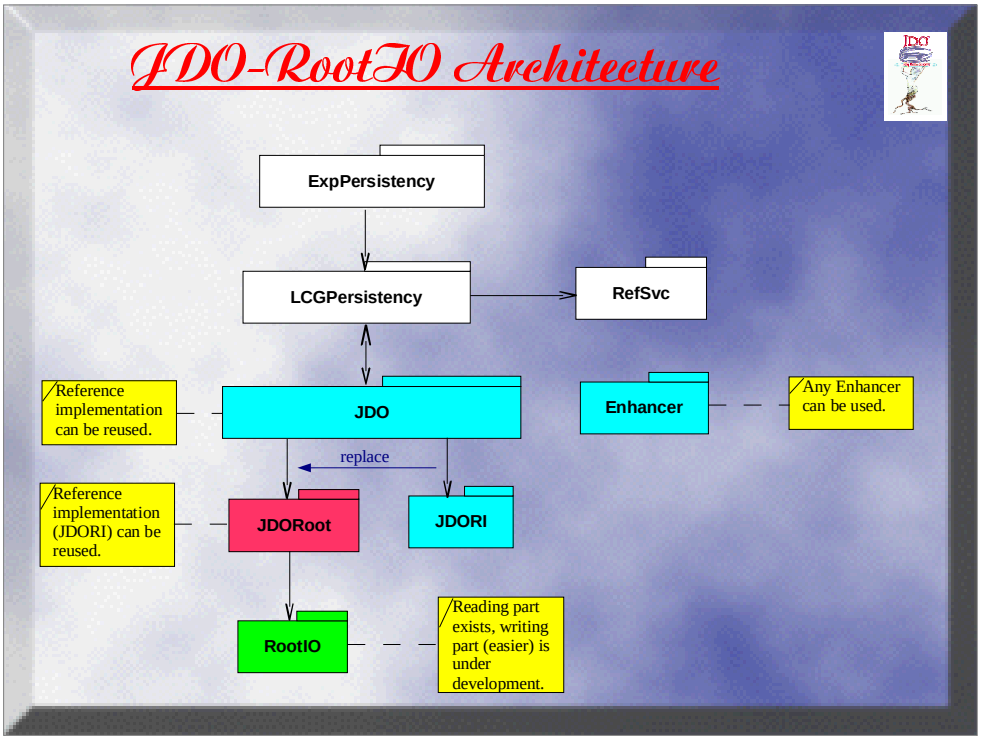
•Worth to look at.

Motivation for JDO-RootIO

- *It satisfies well requirements as defined by RTAG*
- *It profits from the existing mature technologies, which can be reused:*
  - *Root files*
  - *Java RootIO*
  - *JDO API*
  - *Open JDO implementations:*
    - *Reference*
    - *GNU*
  - *Java environment*
- *It gives transparent access to the same data from both Java and C++ at the same time*
- *It offers plurality of persistency technologies (Root files, any RDBS, OODBS, other file-formats,...)*

•JDO itself satisfy all RTAG Requirements except the requirements that implementation should use Root. But this hole can be easily fixed.

•Writing to Root files can be ready in about half year.
•BCEL dynamicaly creates objects in memory according to description read from Root file. The source for those classes can be saved as well.

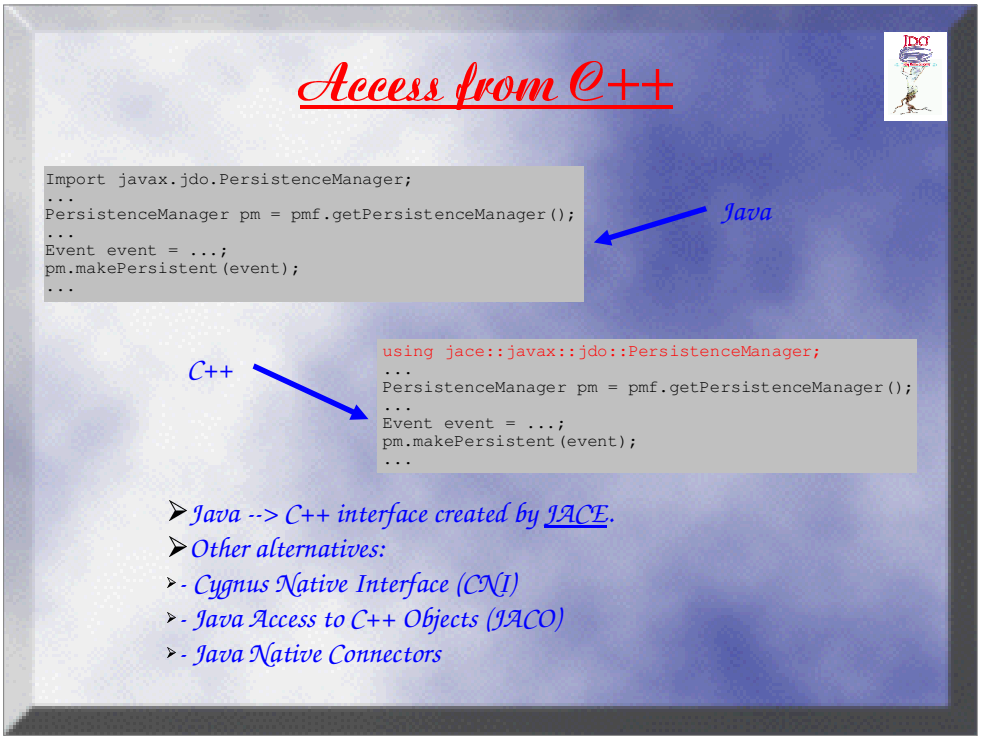


- Enhancer is precisely defined - easy to re-implement. Special treatment of Root files (split mode, ...) can be added.
- Blue: ready (standard components which can be re-used).
- Green: partly ready (reading works well, writing under development).
- Red: to do.
- White: common for all LCG implementations, language neutral: realtional layer, OID,...

RootIO as Java Streaming

- Standard Java Streaming can be easily implemented on top of basic Java RootIO too.
- It would deliver just basic Object IO services.
- It has several advantages:
  - Simple API
  - Same API as Native Java Streams
  - Chaining with other Streams
  - Operation over the Network
  - Connection between Processes
  - Connection to hardware
- Higher level services could be implemented also on top of this interface.
- Compatibility between JDO RootIO and Streaming RootIO (OID, References,...) is desirable.

•Closer to native RootIO implementation, but lacks DB functionality.



•JNI is not difficult, just tedious. It's important to get a tool which automates task of creation of interfaces. There are several candidates, JACE seems to be the best.

- Unlike Java, porting of C++ part of JACE to new platform/compiler/library requires significat effort (one has to compile thousand classes).
- Linux g++ port is under active development.

Relation to RTAG Architecture (1)

- *Standard JDO PersistenceManager can be used. It provides all required functions. Its behaviour can be customised via Properties (text file) of the PersistenceManagerFactory. Several PersistenceManagers can co-exist (connected to different files or even different technologies).*
- *StorageMgt task is performed directly by the Java RootIO.*
- *Basic Transient Cache Management is provided directly by Java itself. Higher level DB-like functions (keys, meta-information passing,...) can be implemented on top using InfoBus or JavaSpaces. JACE interface allows reusing of existing C++ Transient Cache Managers.*

- How it satisfy RTAG Architecture ?

Relation to RTAG Architecture (2)

- *References within one JDO DB (Root file) are handled automaticaly. External References can be handled by PersistenceManagerFactory with connection to standard Relational Service (Grid, JNDI ?). PersistenceManagerFactory creates proper PersistenceManager connected by JDO DB which containes required Object. The address of this JDO DB is obtained from the Relational Service. Dynamic Proxies can be used for transparent handling of the remote References at the language level. Those can also provide additional functionality (caching, lazy-loading, placement,...). Identity managed by Application (i.e. LCG identity) should be compatible with the native C++ OID.*
- *Natural granularity for Placement is JDO DB (Root file), it can be trivially implemented using PersistenceManager Properties. Finer granularity can be implemented as JDO extension.*

- Caching within one JDO DB is provided directly by JDO.
- JDO uses Coollections as hints for data clustering, further level of clustering can be specified in class description XML file.

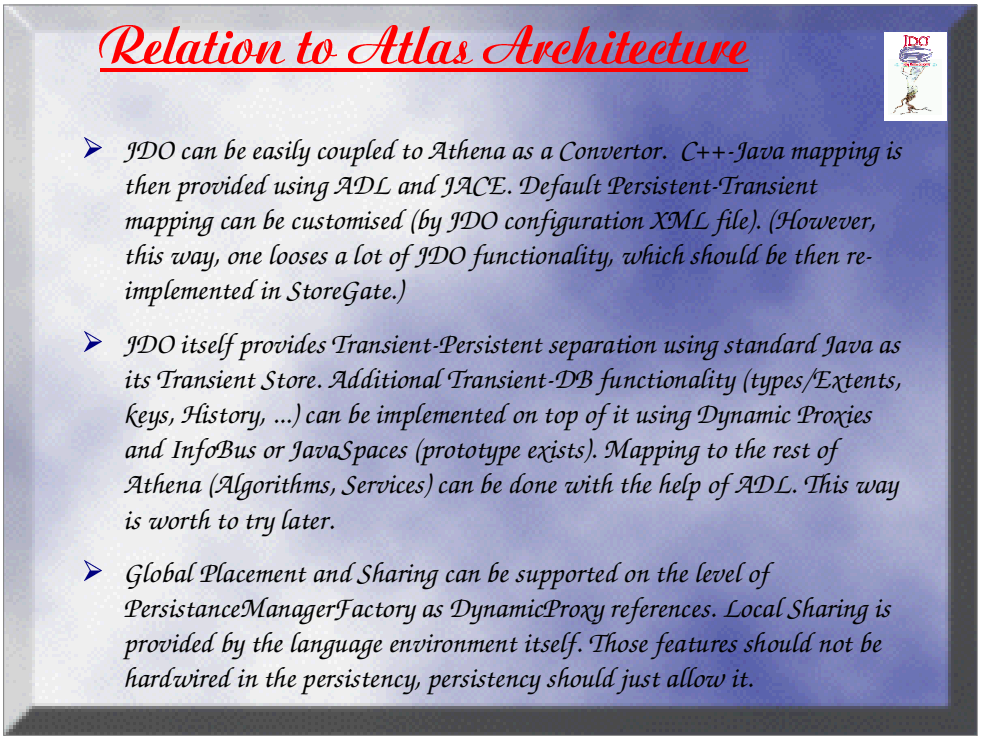
Relation to RTAG Architecture (3)

- Transient Dictionary is not needed thanks to Java Reflection. Persistent-Transient Dictionary is handled by JDO itself (via standard XML file). Default mapping can be to some extend customised. Further level of customisation can be implemented on top of JDO (by Dynamic Proxies).
- JDO fully supports all standard Java Collections. Collections are internaly handled in a special way insuring good performance. High level collection management can be implemented on top.

From RTAG Report (Ch.4.1)

- *Components ... should implement abstract interfaces and be as technology neutral as possible. ...*
- *The interaction ... should happen exclusively through the public and agreed interfaces. ...*
- *... A thin layer to hide the technicalities of such interfaces should be envisaged.*
- *... If implementations already exists providing the required functionality they should be used to provide the initial implementation.*
- *We target C++ as the main programming language, however we should avoid contructions, which makes impossible the migration to existing or future new languages.*
- *... Transient objects whose states will be saved/restored will be compiled and linked without knowledge of any specific persistence technology.*

•Very well satisfied by JDO.



•Interface to Athena (C++) certainly downgrades the functionality of JDO as all Athena has been designed with C++ limitations in mind.
•There are two layers of Transient-Persistent separation (Cnv) in the Simple model: Root-file -- JDO(Cnv) -- Java -- JACE -- C++ -- AthenaCnv -- DataObject. However, the data are copied only once - in AthanaCnv. JDO reads into memory (it doesn't do any additional copy), JACE provides remote access.
•There is an additional mapping work to be done work Objects which are not supported by RootIO, but this work is needed in all implementations.
•PersistanceManagerFactory creates PersistantManager working on particular instance of JDO DB. This can be used both to place Collections as desired and to find them (in collaboration with Navigation/Relational layer).
•JACE uses as the source of it description Java Class, which is in this design created (staticaly on dynamicaly, possibly includin JDO-customised mapping) from the the exsisting Root file. Athena, however uses ADL to create C++ (and other) representations. This doesn't pose any problem as long as all sources (i.e. ADL and objects inside Root file) are consistent.

•This project requires much less manpower that the all-in-C++ project to deliver at least equivalent functionality in both Java and C++.
•The project is realistic, most components are already fully functional.
•The required work consist mainly from glueing all components together, interfacing them into common LCG Framework (and through that into experiments' Frameworks) and replacing existing components with others.

Conclusion

- *Proposed solution delivers at least equivalent functionality as the mainstream (fully C++, Root-based) one thanks to reuse of mature language and products.*
- *It satisfies all Atlas and RTAG requirements in a modular way (many components can be replaced by equivalent components without loss of consistency).*
- *It requires much less manpower thanks to massive reuse of existing code and concepts (many features which have to be implemented in the mainstream solution already work here).*
- *It works at the same time in both Java and C++ environments, with the same API. C++ proxies can be build completely automaticaly, no other special C++ processing (dictionary, pre-processor, ...) is needed. It is clear, however, that C++ API is less functional than Java one.*
- *It always uses widely accepted standards and Open or HEP products.*
- *It allows using of wide range of persistent technologies thanks to modular reuse of abstract API.*
- *Work can start immediately as all components have existing alternatives so that the whole chain is functional thanks to modularity of the architecture.*

Links

- *JDO:*
  http://www.jdocentral.com
- *JDO Reference Implementation:*
  http://access1.sun.com/jdo/
- *Sparrow/ObjectBridge:*
  http://sparrowdb.org/
  http://objectbridge.sourceforge.net/
- *Root:*
  http://root.cern.ch
- *Java Root IO:*
  http://java.freehep.org/lib/freehep/doc/root/index.shtml
- *JACE:*
  http://reyelts.dyndns.org:8080/jace/index.html
- *This Presentation:*
  http://home.cern.ch/~hrivnac/2002/May/JDO/