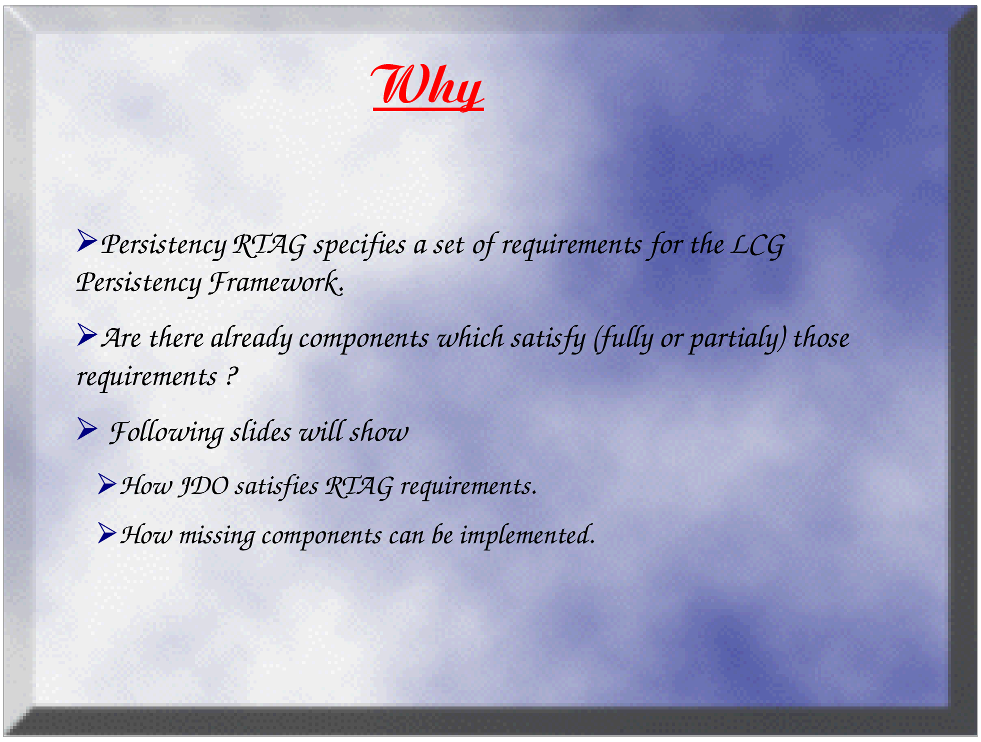
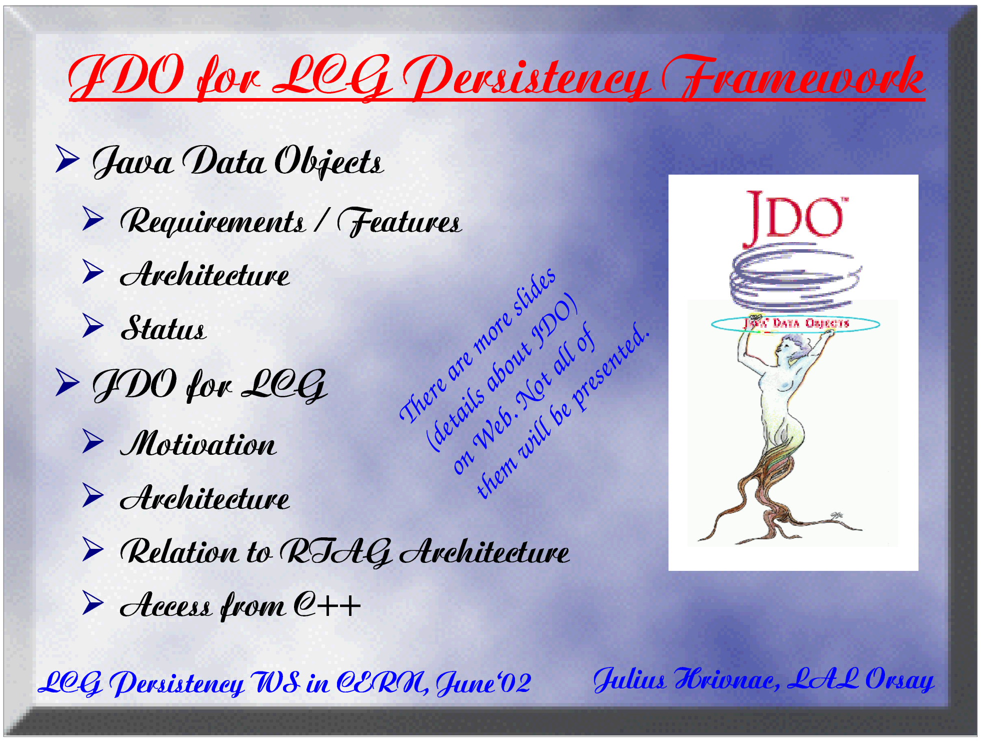


Why

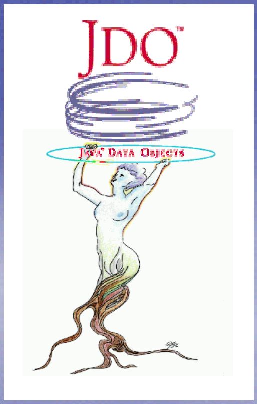
- Persistency RTAG specifies a set of requirements for the LCG Persistency Framework.
- Are there already components which satisfy (fully or partialy) those requirements ?
- Following slides will show
  - How JDO satisfies RTAG requirements.
  - How missing components can be implemented.



JDO for LCG Persistency Framework

- Java Data Objects
  - Requirements / Features
  - Architecture
  - Status
- JDO for LCG
  - Motivation
  - Architecture
  - Relation to RTAG Architecture
  - Access from C++

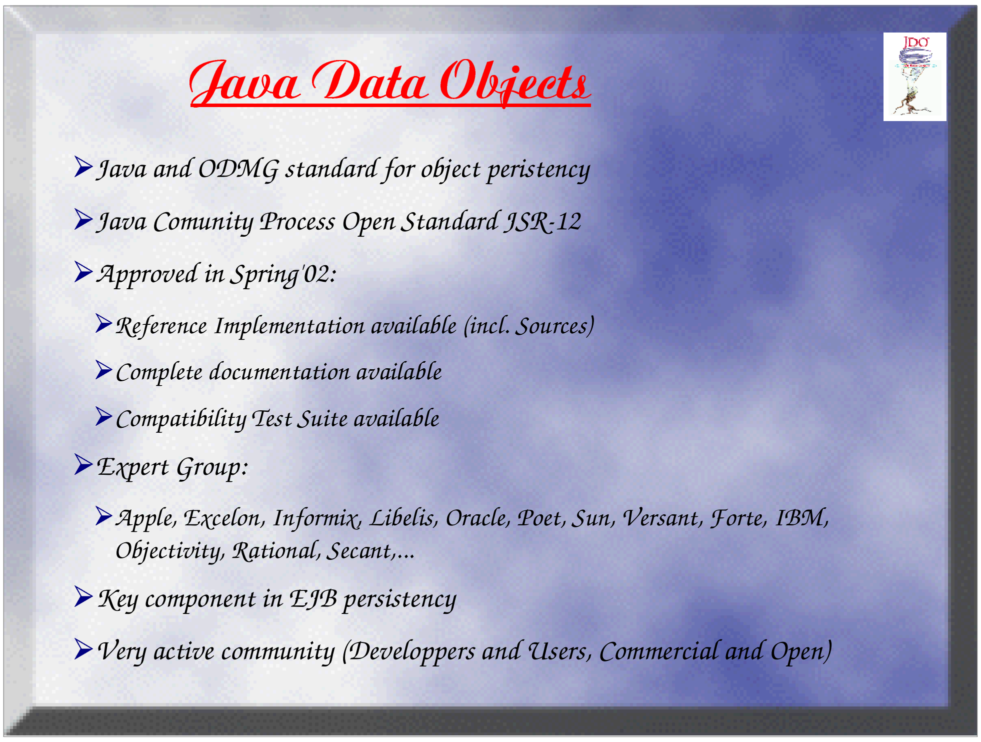
There are more slides (details about JDO) on Web. Not all of them will be presented.



LCG Persistency WS in CERN, June'02

Julius Hrivnac, LAL Orsay

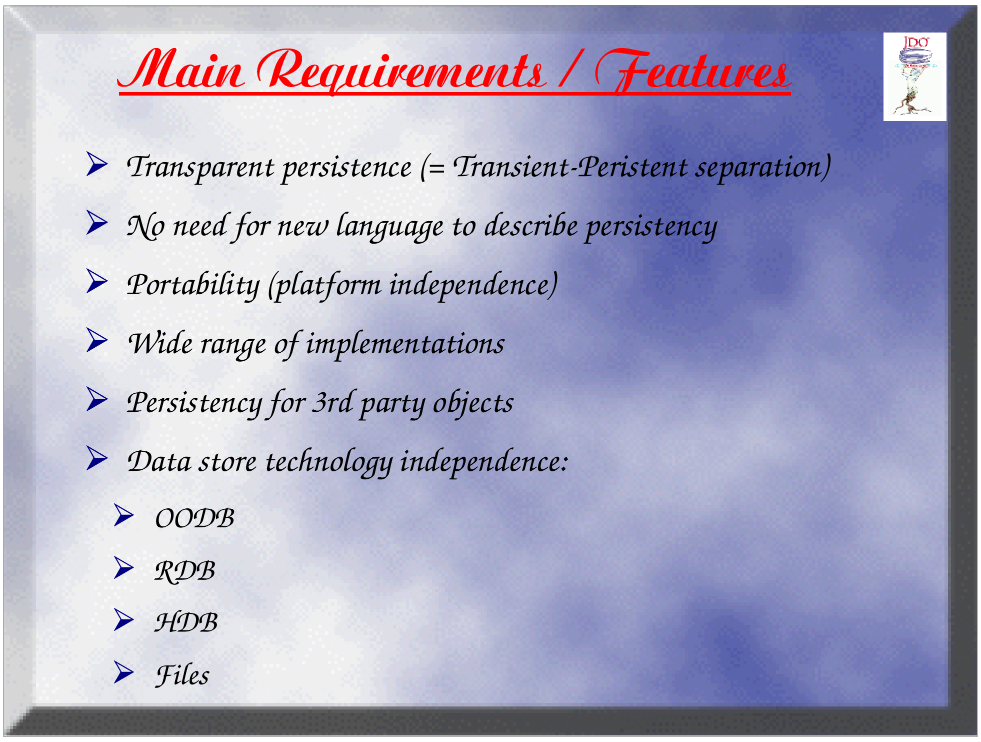
•Concrete realistic proposal how to implement (Root) Persistency Service for both Java and C++ using JDO.
•In the Atlas Java Note (almost two years ago), we wrote that persistency is not resolved in Java. It is no more true. Java now offers persistency APIs superior to those of C++ (JDO, JDBS, jSQL, 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.



Java Data Objects

- Java and ODMG standard for object peristency
- Java Comunity Process Open Standard JSR-12
- Approved in Spring'02:
  - Reference Implementation available (incl. Sources)
  - Complete documentation available
  - Compatibility Test Suite available
- Expert Group:
  - Apple, Excelon, Informix, Libelis, Oracle, Poet, Sun, Versant, Forte, IBM, Objectivity, Rational, Secant,...
- Key component in EJB persistency
- Very active community (Developpers and Users, Commercial and Open)

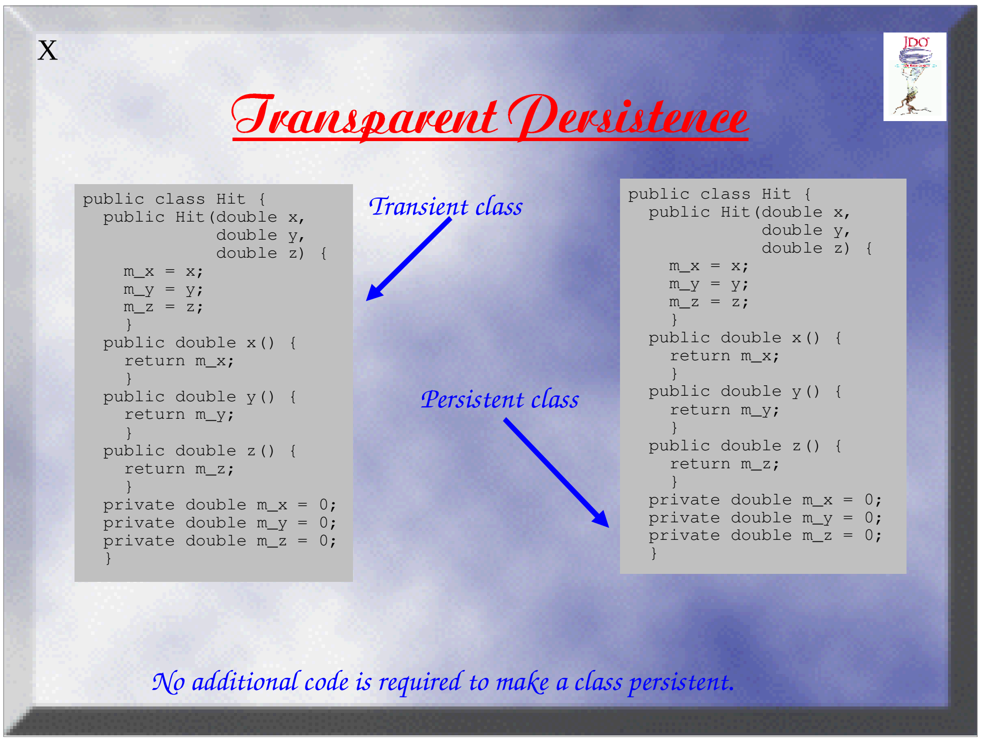
•Each Java standard should have Reference Implementation and Test Suite.
•Very active developpers community, many existing activities and products (both commercial and Free).



Main Requirements / Features

- Transparent persistence (= Transient-Peristent separation)
- No need for new language to describe persistency
- Portability (platform independence)
- Wide range of implementations
- Persistency for 3rd party objects
- Data store technology independence:
  - OODB
  - RDB
  - HDB
  - Files

- Transparent Persistence = Transien-Persistent Separation.



X

Transparent Persistence

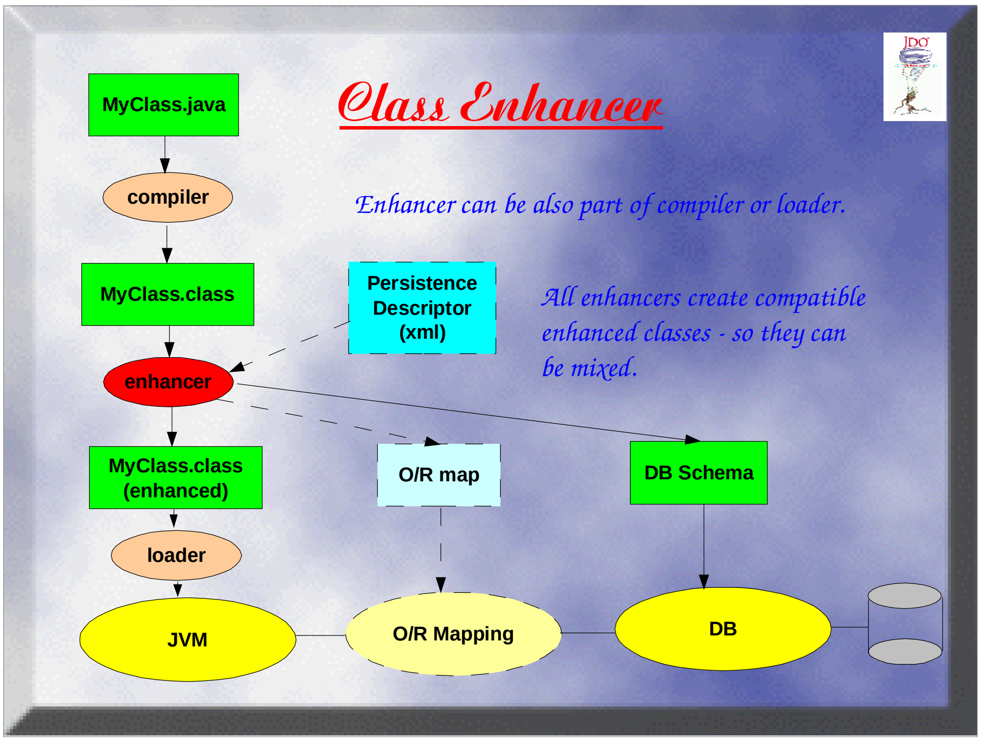
Transient class

```
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;
  }
```

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.

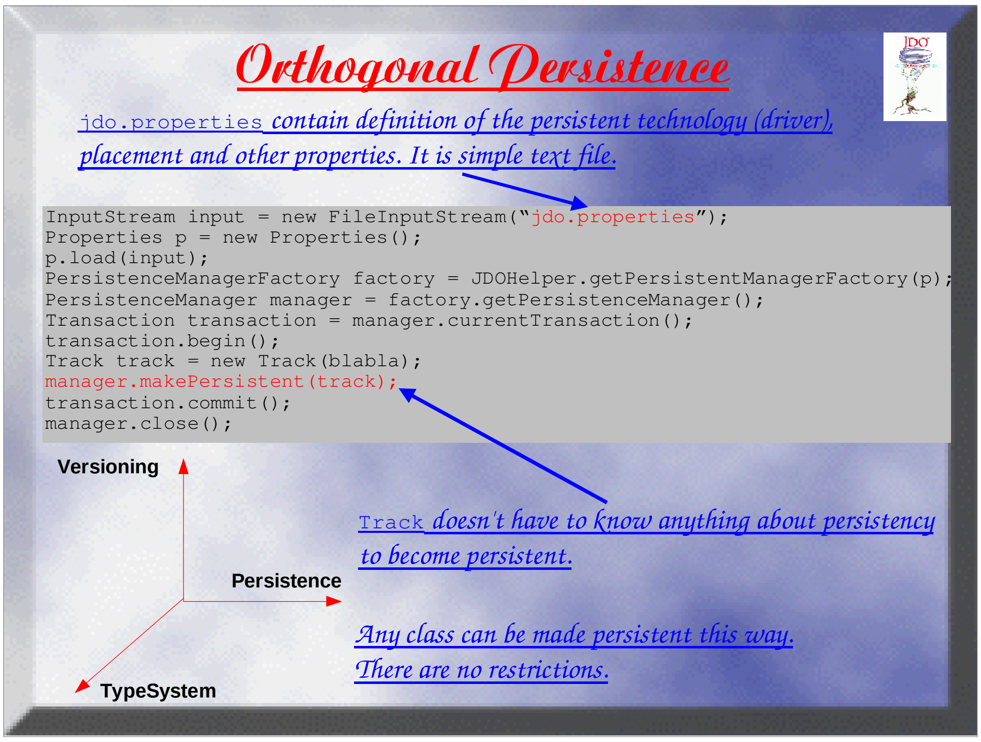


Class Enhancer

Enhancer can be also part of compiler or loader.

All enhancers create compatible enhanced classes - so they can be mixed.

•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,...



Orthogonal Persistence

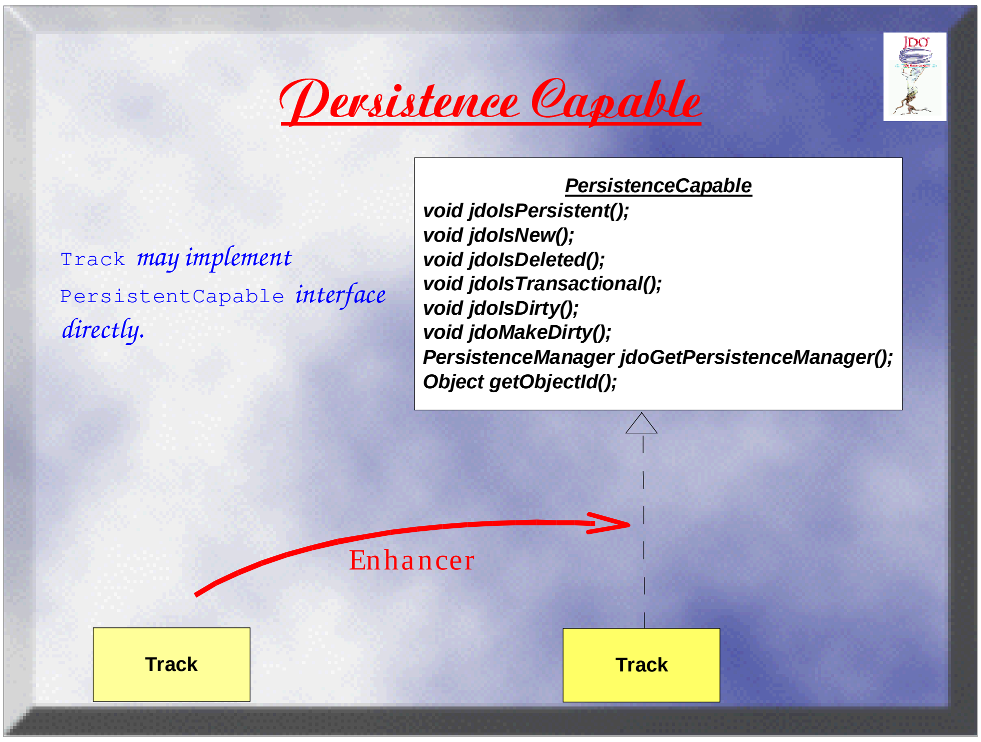
`jdo.properties` *contain definition of the persistent technology (driver), placement and other properties. It is simple text file.*

```
InputStream input = new FileInputStream(“jdo.properties”);
Properties p = new Properties();
p.load(input);
PersistenceManagerFactory factory = JDOHelper.getPersistentManagerFactory(p);
PersistenceManager manager = factory.getPersistenceManager();
Transaction transaction = manager.currentTransaction();
transaction.begin();
Track track = new Track(blabla);
manager.makePersistent(track);
transaction.commit();
manager.close();
```

`Track` *doesn't have to know anything about persistency to become persistent.*

Any class can be made persistent this way. There are no restrictions.

•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.

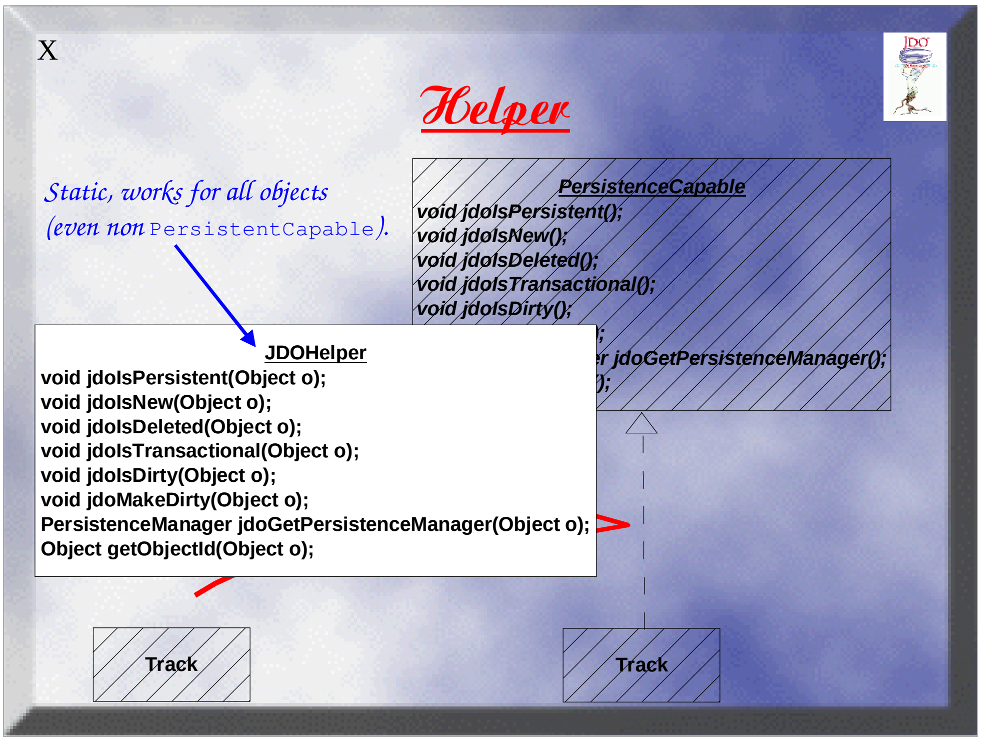


Persistence Capable

Track *may implement* PersistentCapable *interface directly.*

```
PersistenceCapable
void jdoIsPersistent();
void jdoIsNew();
void jdoIsDeleted();
void jdoIsTransactional();
void jdoIsDirty();
void jdoMakeDirty();
PersistenceManager jdoGetPersistenceManager();
Object getObjectId();
```

•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.



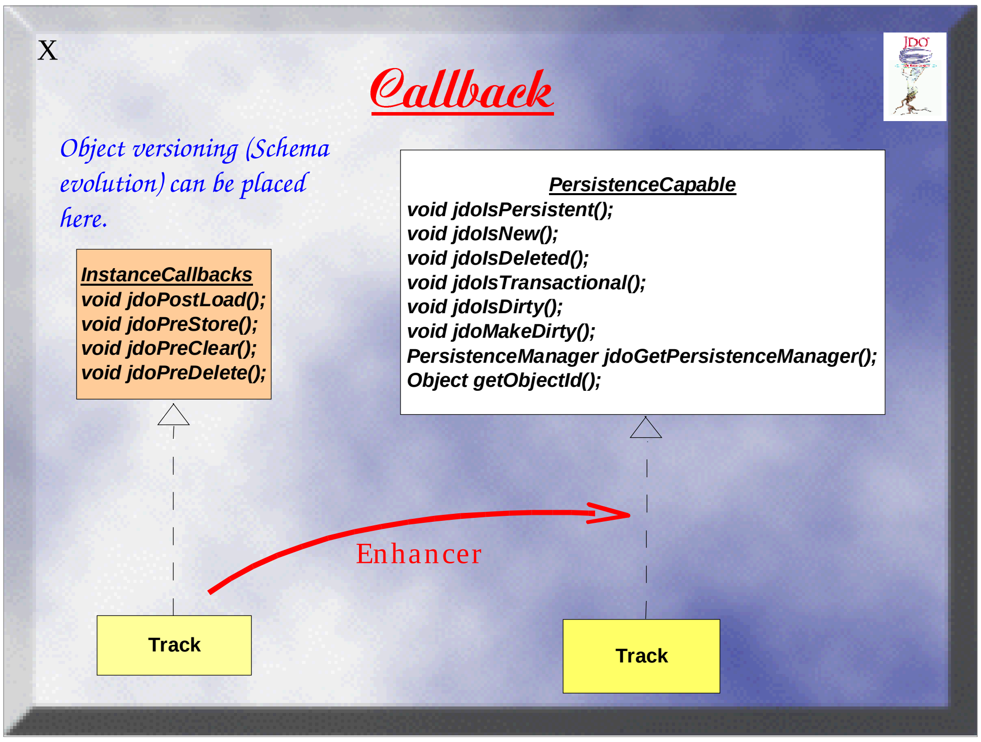
X

Helper

Static, works for all objects (even non `PersistentCapable`).

PersistenceCapable
void jdoIsPersistent();
void jdoIsNew();
void jdoIsDeleted();
void jdoIsTransactional();
void jdoIsDirty();
r jdoGetPersistenceManager();
JDOHelper
void jdoIsPersistent(Object o);
void jdoIsNew(Object o);
void jdoIsDeleted(Object o);
void jdoIsTransactional(Object o);
void jdoIsDirty(Object o);
void jdoMakeDirty(Object o);
PersistenceManager jdoGetPersistenceManager(Object o);
Object getObjectId(Object o);
Track
Track

•Helper is the prefered way of interaction with JDO.

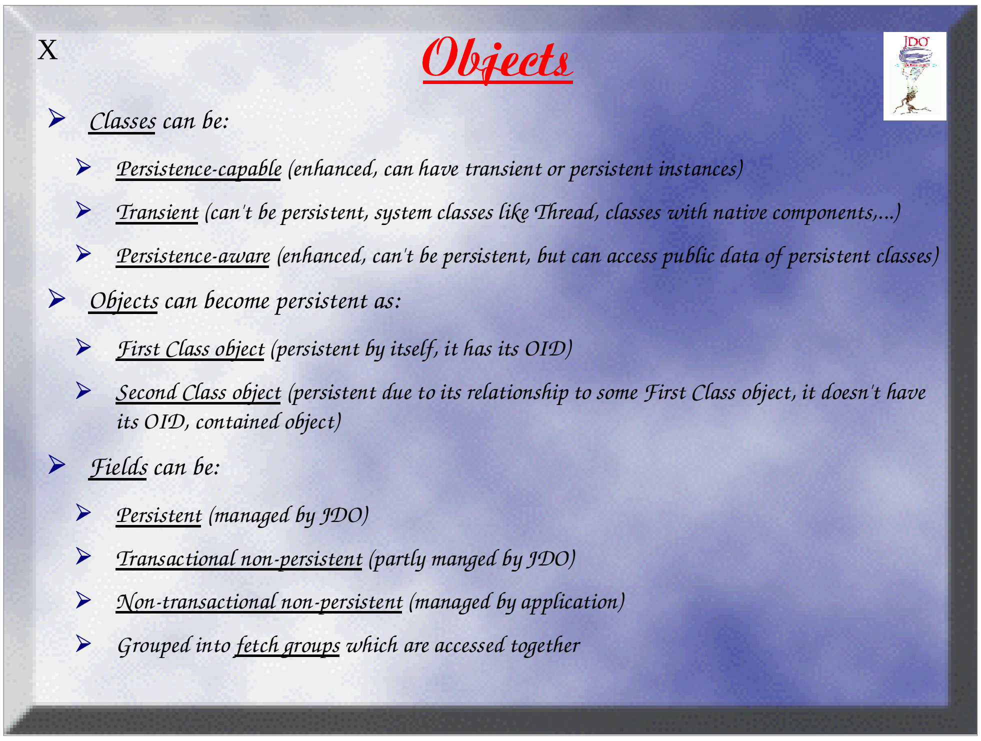


X

Callback

Object versioning (Schema evolution) can be placed here.

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



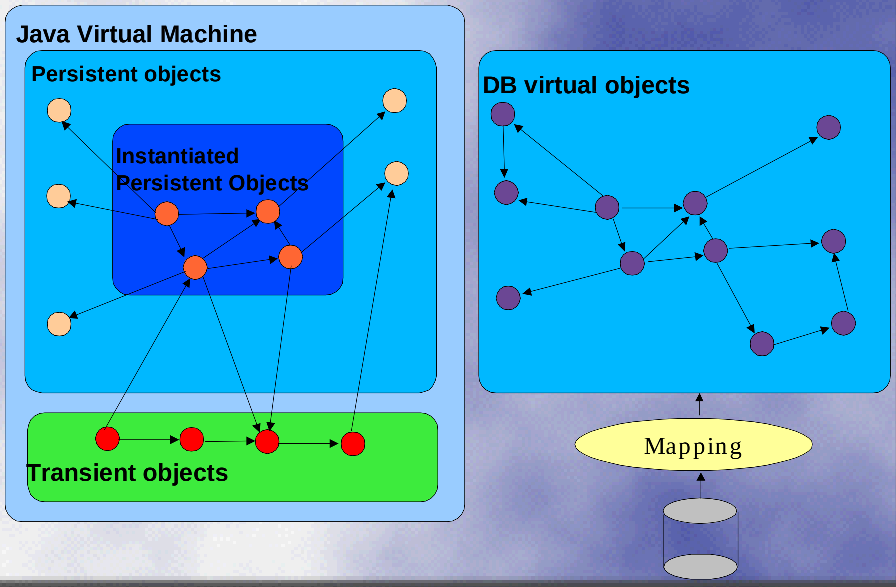
X

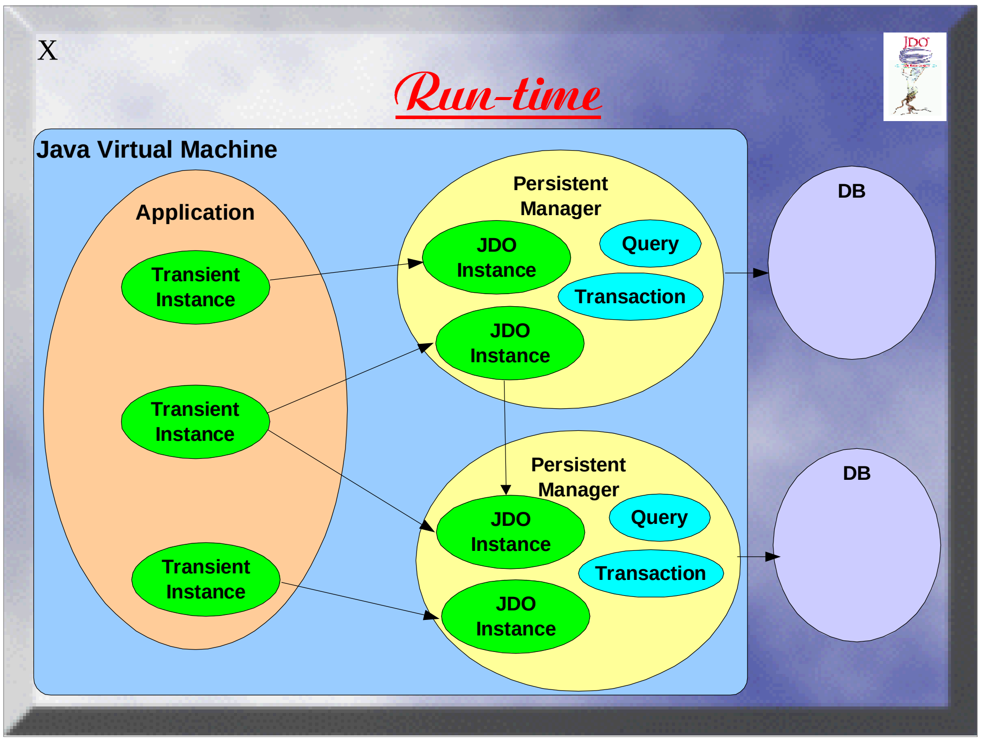
Objects

- 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

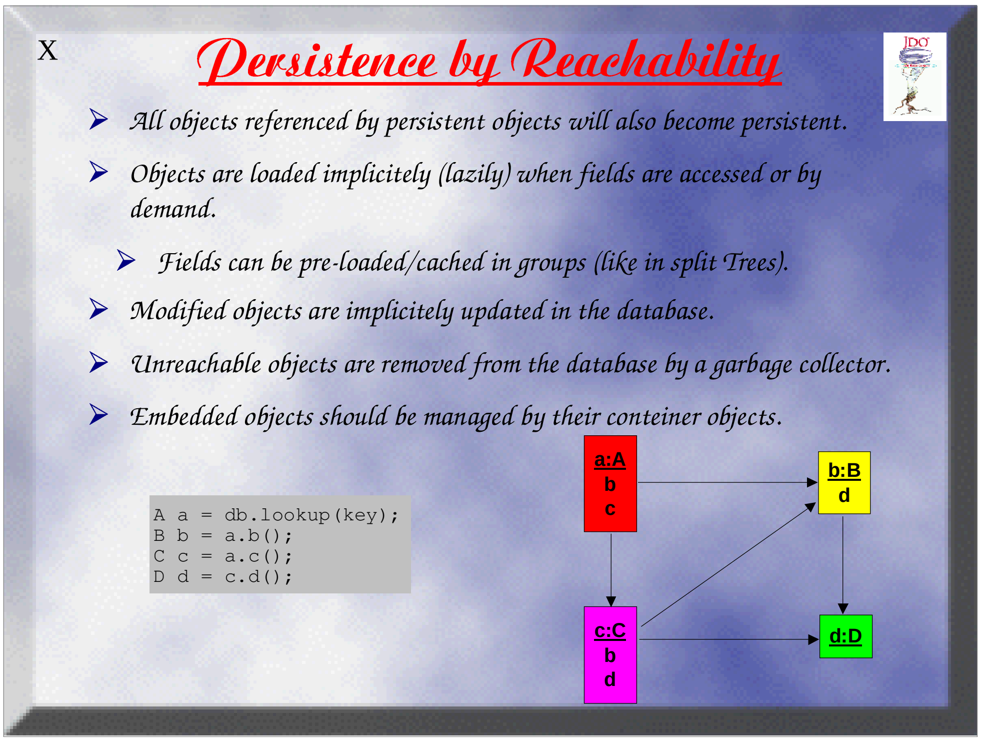
X

Object Model





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

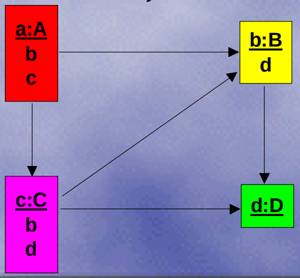


X

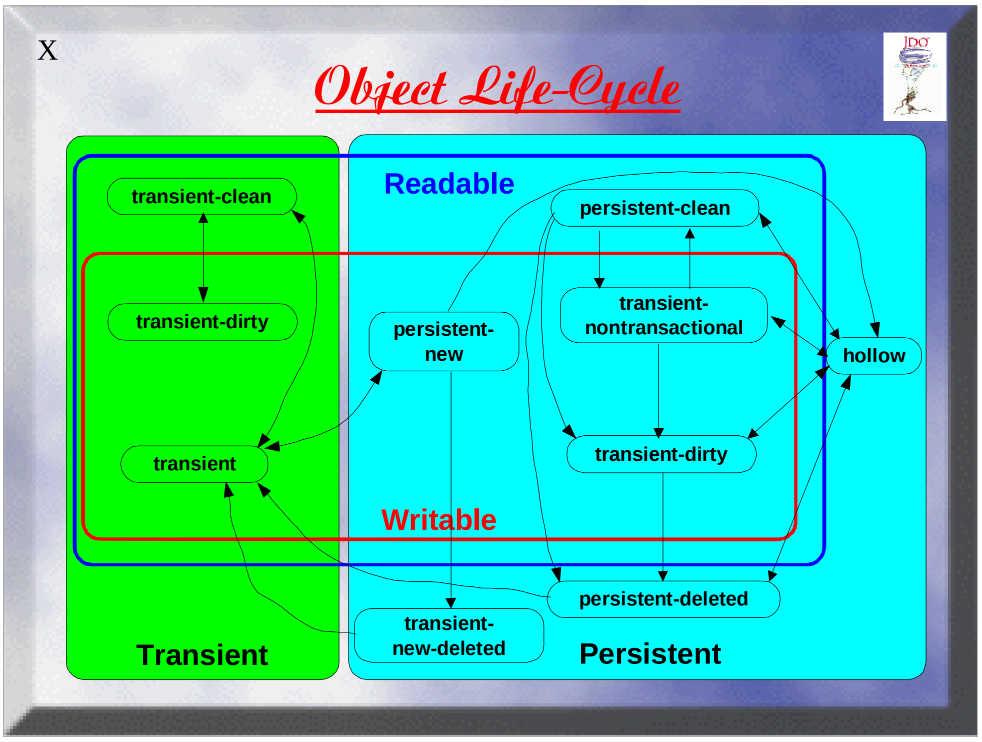
Persistence by Reachability

- 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.

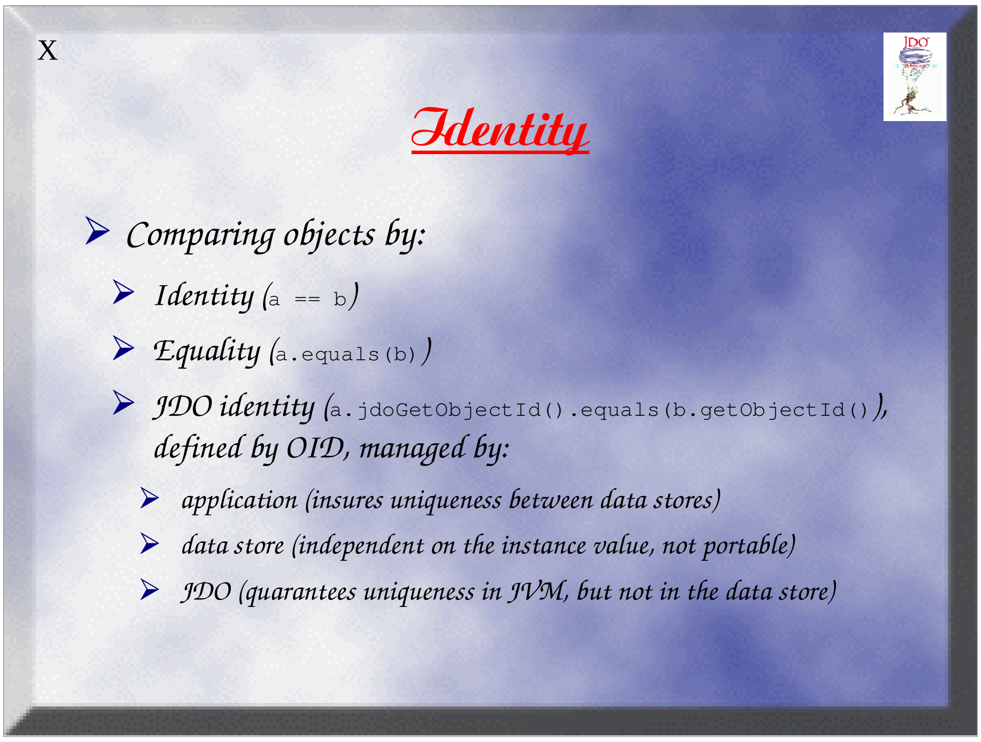
```
A a = db.lookup(key);
B b = a.b();
C c = a.c();
D d = c.d();
```



•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

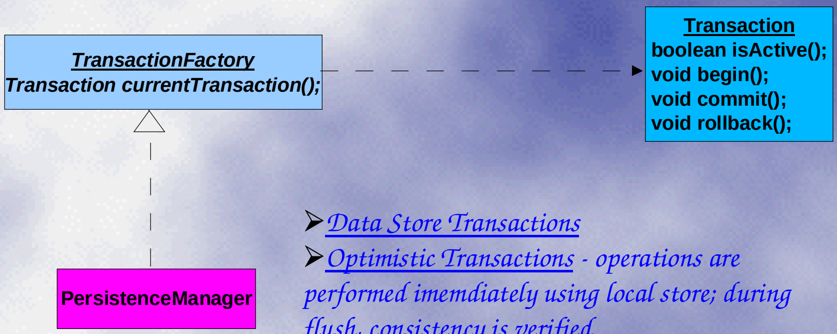
Identity

- Comparing objects by:
  - Identity (`a == b`)
  - Equality (`a.equals(b)`)
  - JDO identity (`a.jdoGetObjectId().equals(b.getObjectId()`), defined by OID, managed by:
    - application (insures uniqueness between data stores)
    - data store (independent on the instance value, not portable)
    - JDO (quarantees uniqueness in JVM, but not in the data store)

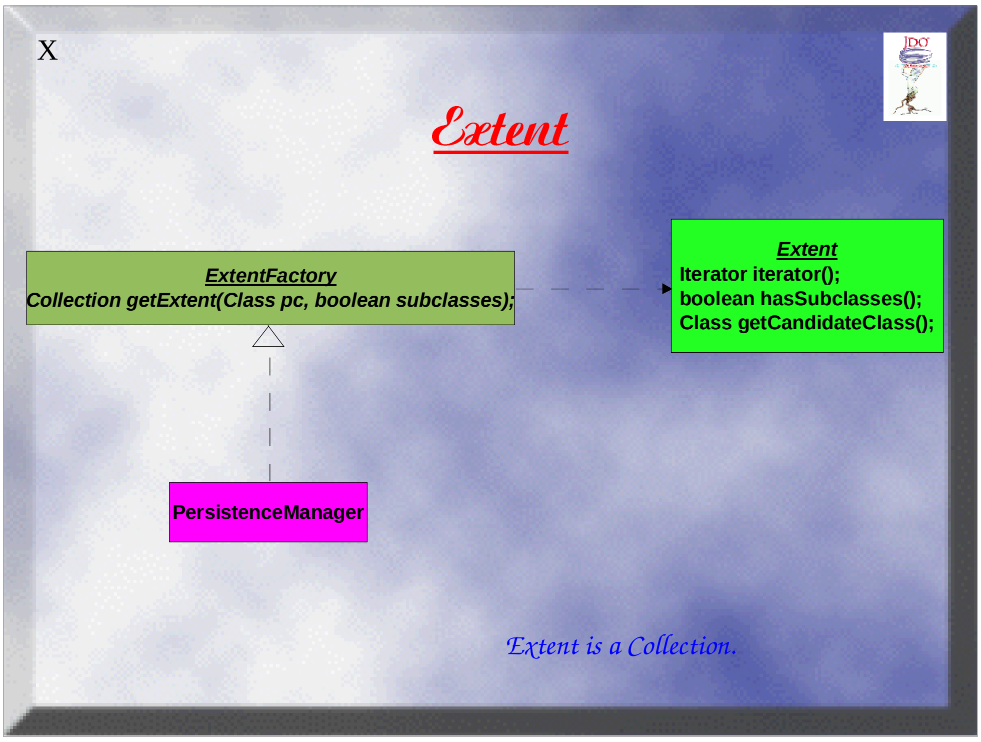
•Identity and Equality are standard Java concepts.
•To insure universal navigation, application identity should be used.

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*

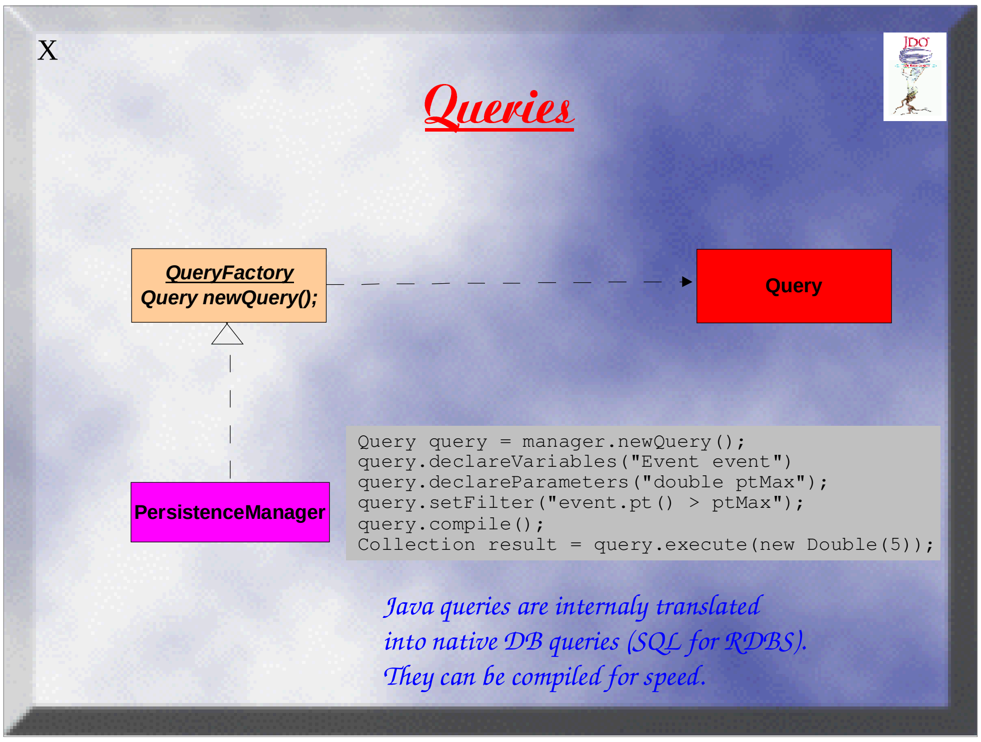


X

Extent

Extent is a Collection.

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

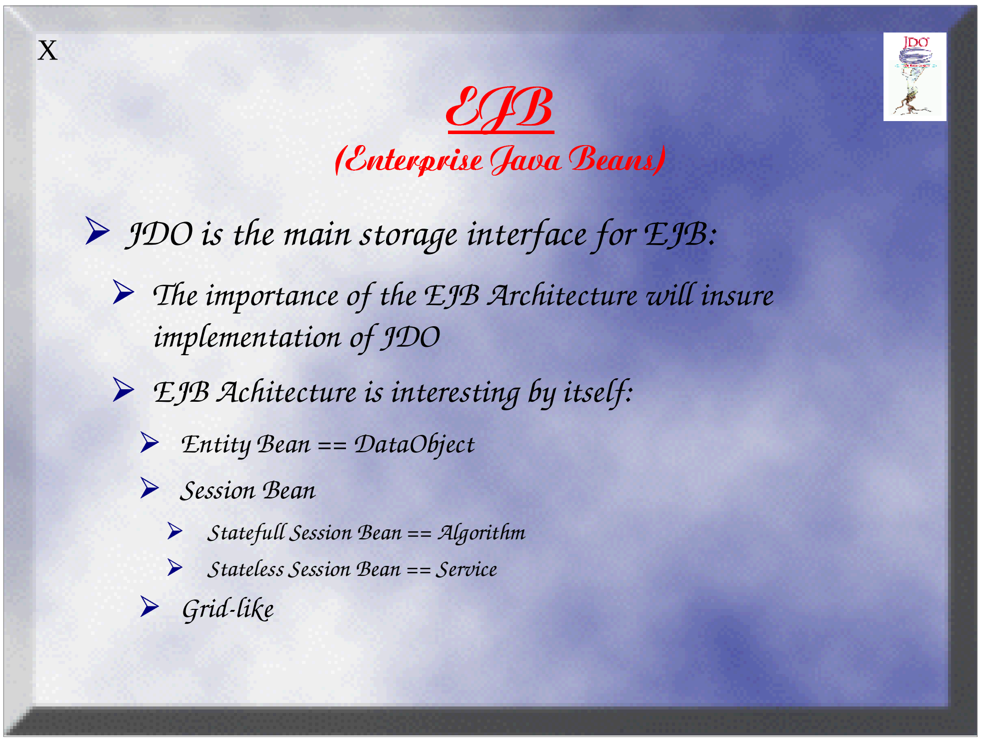

X
JDO
QueryFactory
Query newQuery();
Query
PersistenceManager

Queries

```
Query query = manager.newQuery();
query.declareVariables("Event event")
query.declareParameters("double ptMax");
query.setFilter("event.pt() > ptMax");
query.compile();
Collection result = query.execute(new Double(5));
```

Java queries are internaly translated into native DB queries (SQL for RDBS). They can be compiled for speed.

•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.

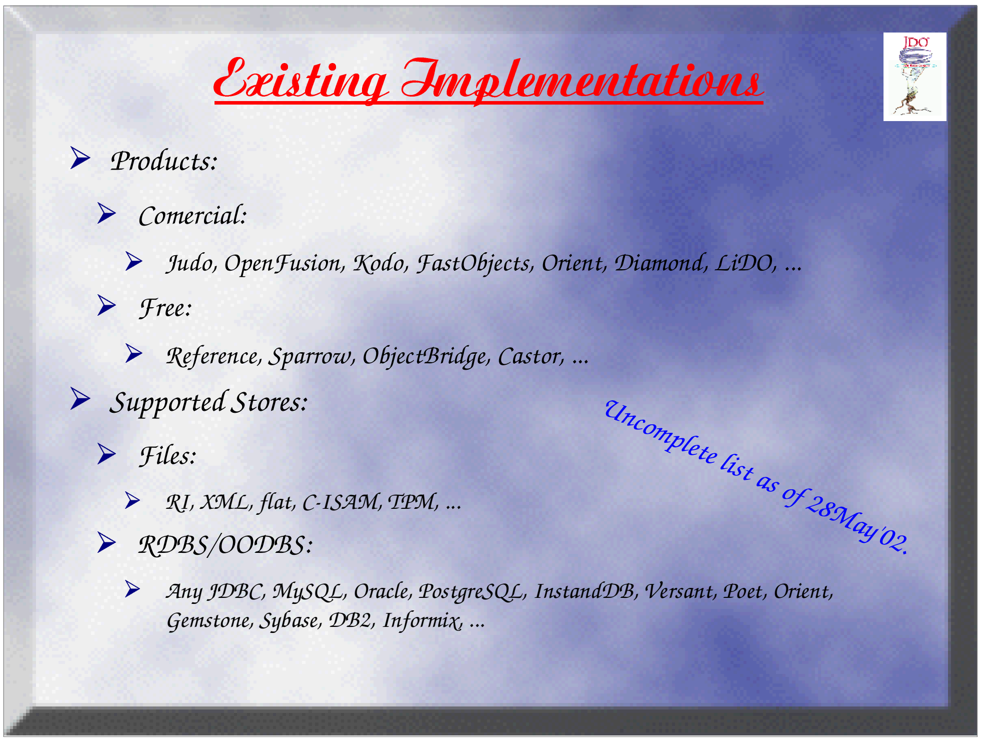


X

EJB
(Enterprise Java Beans)

- JDO is the main storage interface for EJB:
  - The importance of the EJB Architecture will insure implementation of JDO
  - EJB Achitecture is interesting by itself:
    - Entity Bean == DataObject
    - Session Bean
      - Statefull Session Bean == Algorithm
      - Stateless Session Bean == Service
    - Grid-like

•Worth to look at.

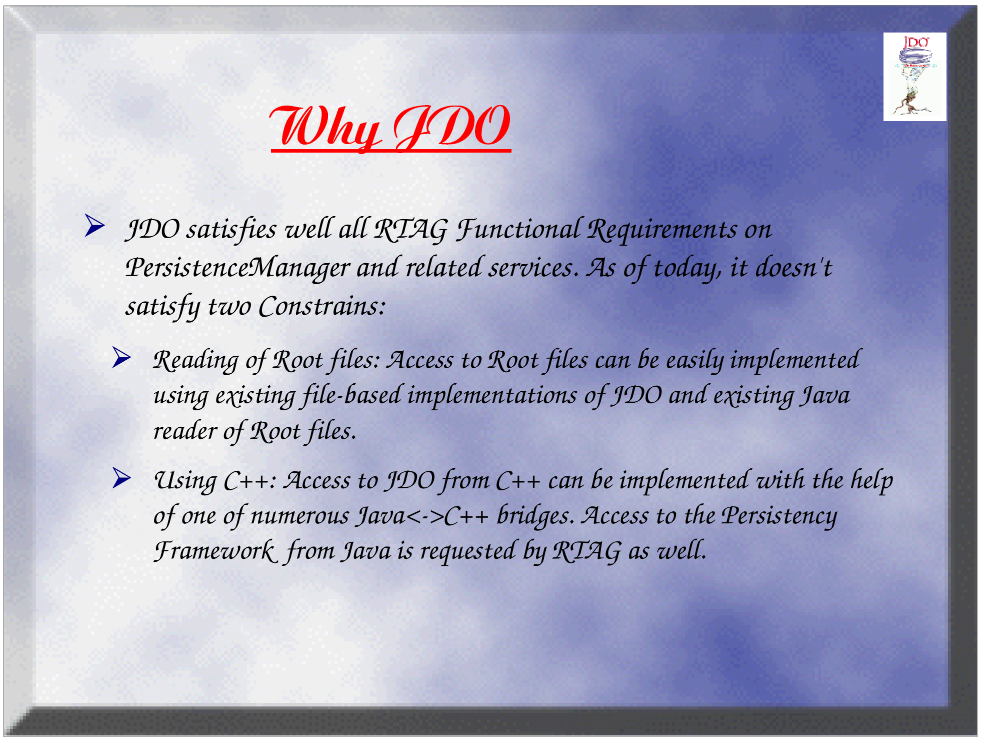


Existing Implementations

- Products:
  - Comercial:
    - Judo, OpenFusion, Kodo, FastObjects, Orient, Diamond, LiDO, ...
  - Free:
    - Reference, Sparrow, ObjectBridge, Castor, ...
- Supported Stores:
  - Files:
    - RI, XML, flat, C-ISAM, TPM, ...
  - RDBS/OODBS:
    - Any JDBC, MySQL, Oracle, PostgreSQL, InstandDB, Versant, Poet, Orient, Gemstone, Sybase, DB2, Informix, ...

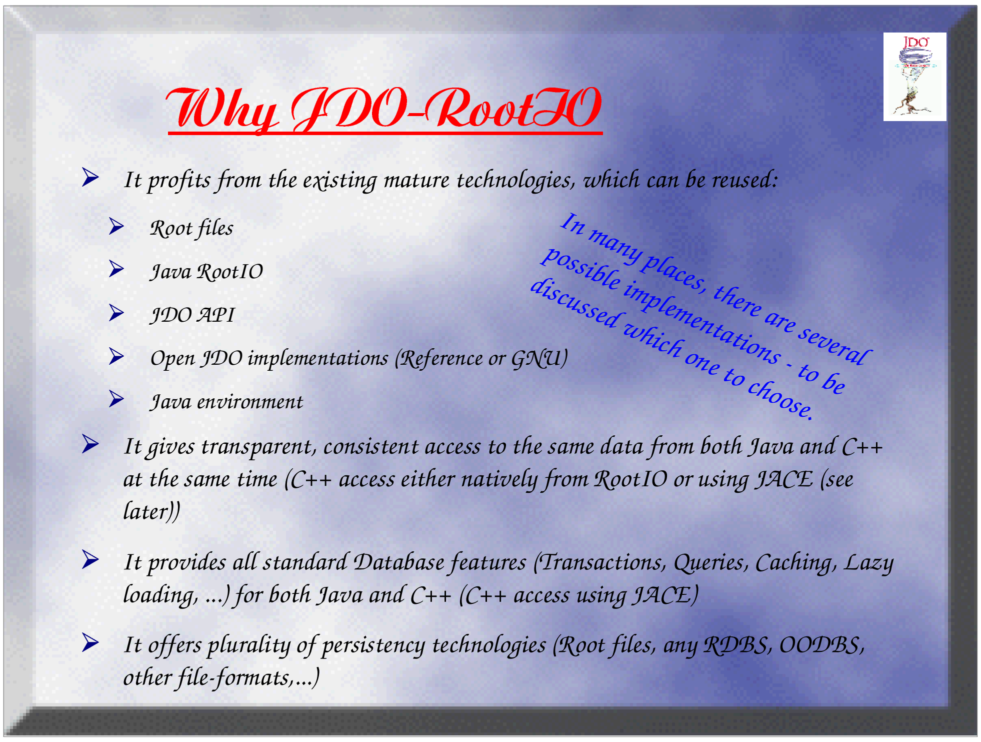
Uncomplete list as of 28May'02.

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



Why JDO

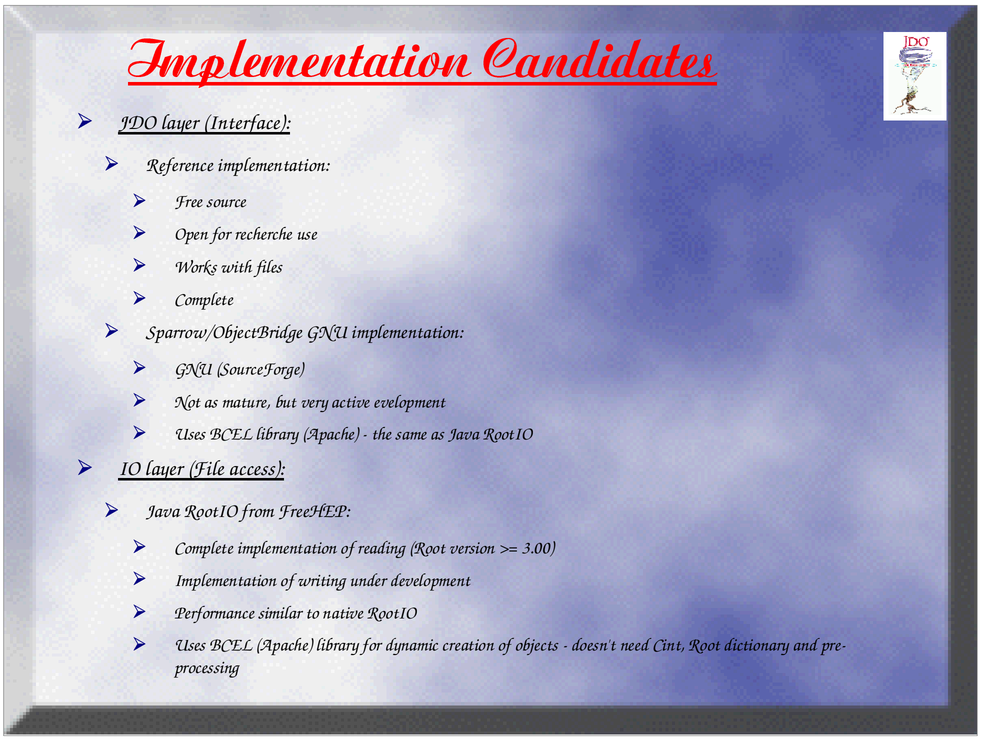
- JDO satisfies well all RTAG Functional Requirements on PersistenceManager and related services. As of today, it doesn't satisfy two Constrains:
  - Reading of Root files: Access to Root files can be easily implemented using existing file-based implementations of JDO and existing Java reader of Root files.
  - Using C++: Access to JDO from C++ can be implemented with the help of one of numerous Java<->C++ bridges. Access to the Persistency Framework from Java is requested by RTAG as well.



Why JDO-RootIO

- It profits from the existing mature technologies, which can be reused:
  - Root files
  - Java RootIO
  - JDO API
  - Open JDO implementations (Reference or GNU)
  - Java environment
- It gives transparent, consistent access to the same data from both Java and C++ at the same time (C++ access either natively from RootIO or using JACE (see later))
- It provides all standard Database features (Transactions, Queries, Caching, Lazy loading, ...) for both Java and C++ (C++ access using JACE)
- It offers plurality of persistency technologies (Root files, any RDBS, OODBS, other file-formats,...)

In many places, there are several possible implementations - to be discussed which one to choose.

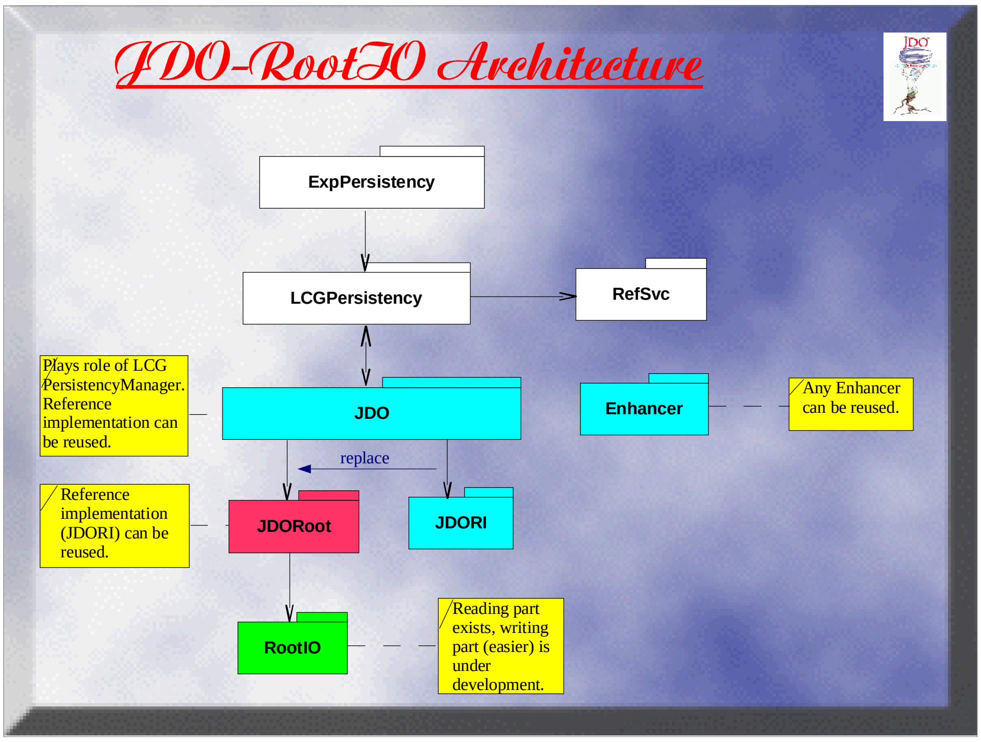


Implementation Candidates

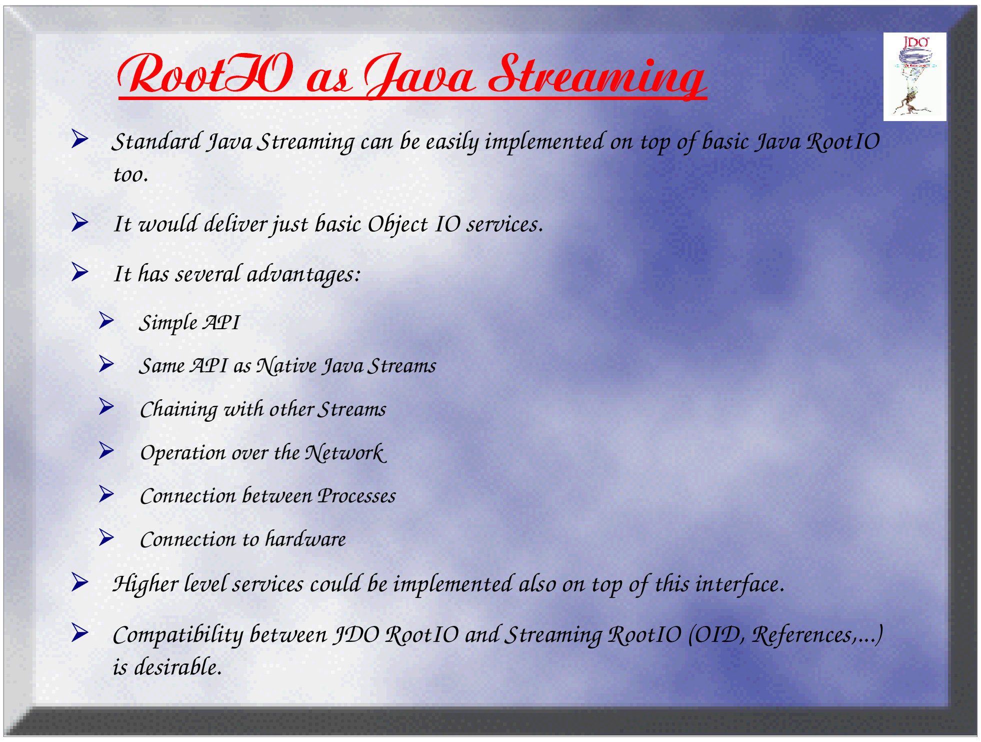
- JDO layer (Interface):
  - Reference implementation:
    - Free source
    - Open for recherche use
    - Works with files
    - Complete
  - Sparrow/ObjectBridge GNU implementation:
    - GNU (SourceForge)
    - Not as mature, but very active evelopment
    - Uses BCEL library (Apache) - the same as Java RootIO
- IO layer (File access):
  - Java RootIO from FreeHEP:
    - Complete implementation of reading (Root version >= 3.00)
    - Implementation of writing under development
    - Performance similar to native RootIO
    - Uses BCEL (Apache) library for dynamic creation of objects - doesn't need Cint, Root dictionary and pre-processing

•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.

JDO-RootIO Architecture


ExpPersistency
LCGPersistency
RefSvc
Plays role of LCG PersistencyManager. Reference implementation can be reused.
JDO
Enhancer
Any Enhancer can be reused.
replace
Reference implementation (JDORI) can be reused.
JDORoot
JDORI
RootIO
Reading part exists, writing part (easier) is under development.

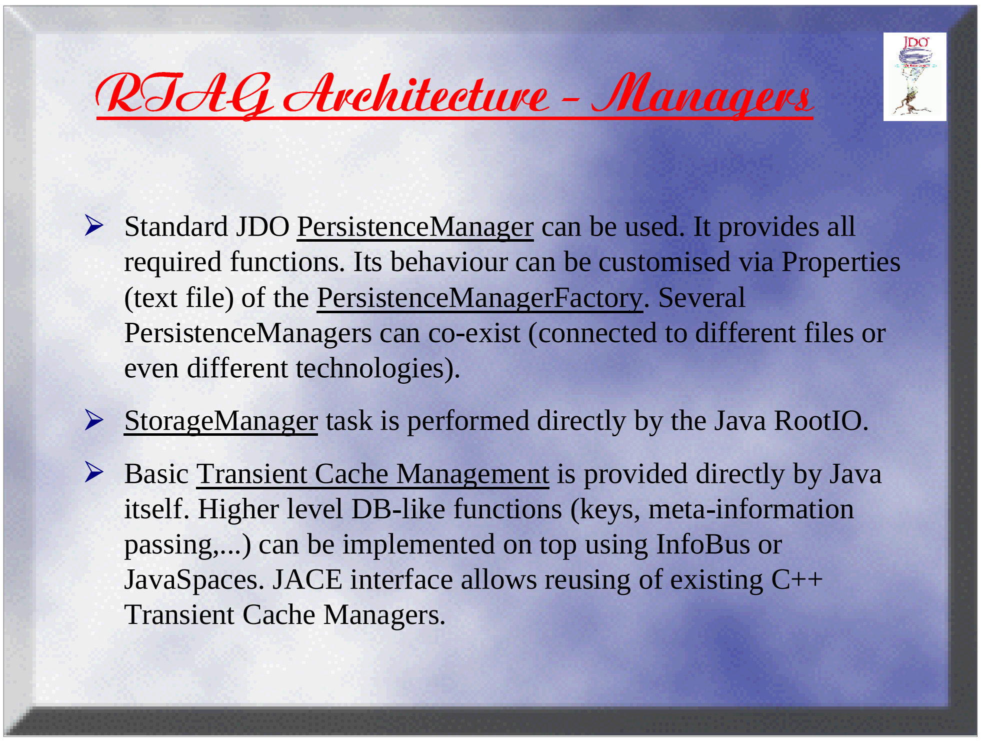
•Enhancer is precisely defined - easy to re-implement. Special treatment of Root files (split mode, ...) can be added.
•Blue: ready (standard components which can be re-used).
•Green: partly ready (reading works well, writing under development).
•Red: to do.
•White: common for all LCG implementations,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

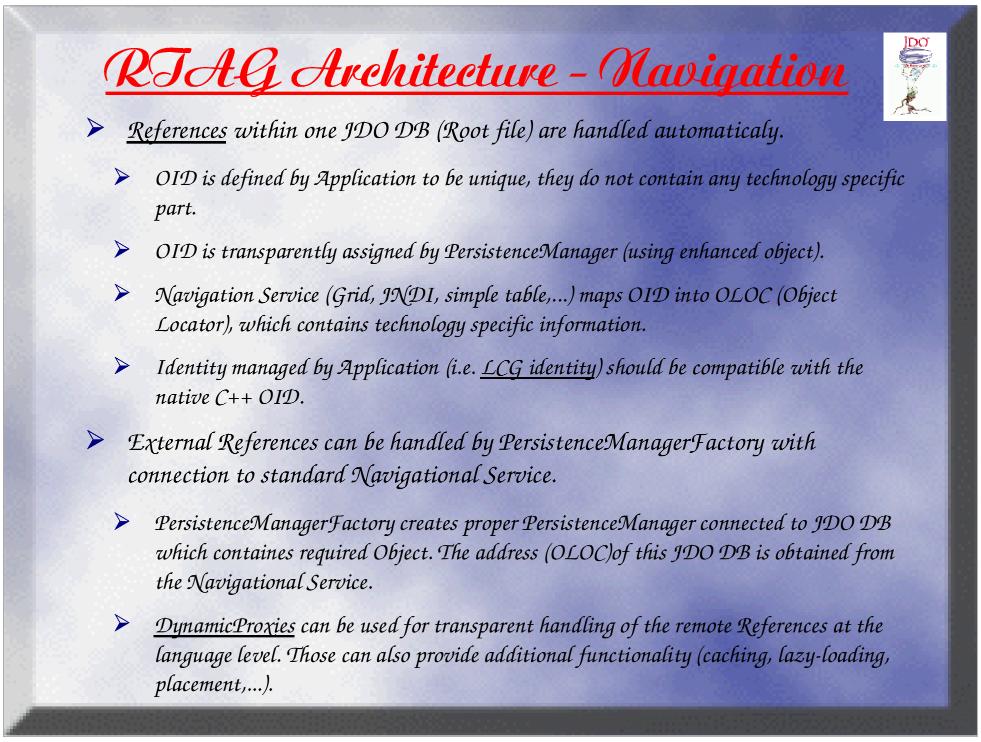
•Closer to native RootIO implementation, but lacks DB functionality.



RTAG Architecture - Managers

- 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).
- StorageManager 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.

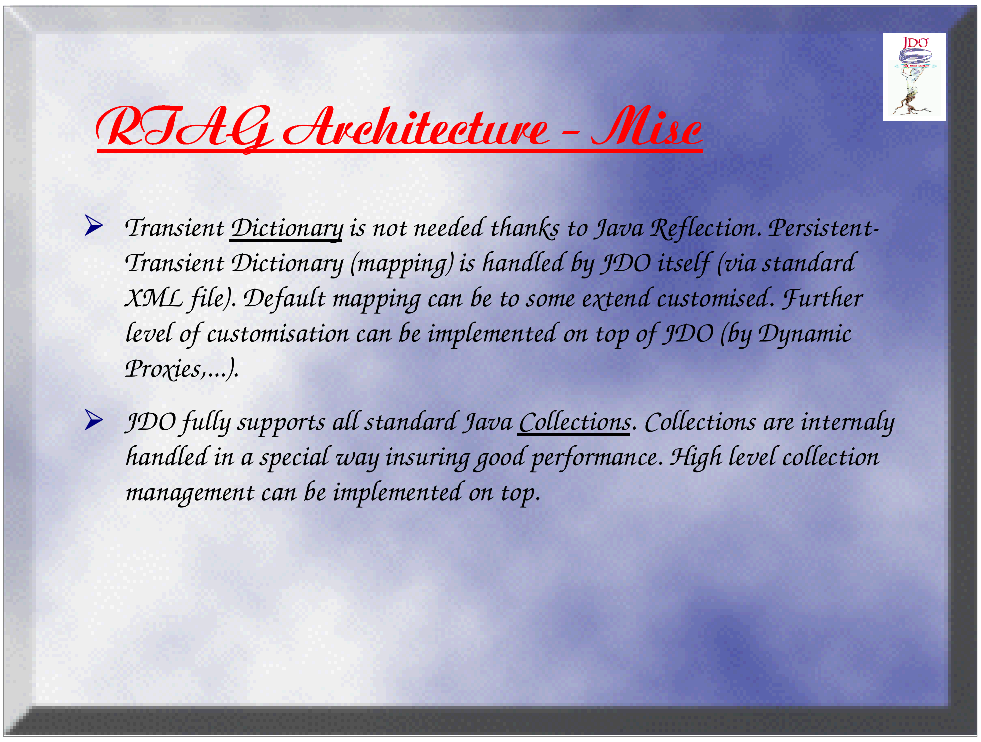
•How it satisfy RTAG Architecture ?
•Transient Cache Manager is not part of this project, but it should be possible to use it.



RTAG Architecture - Navigation

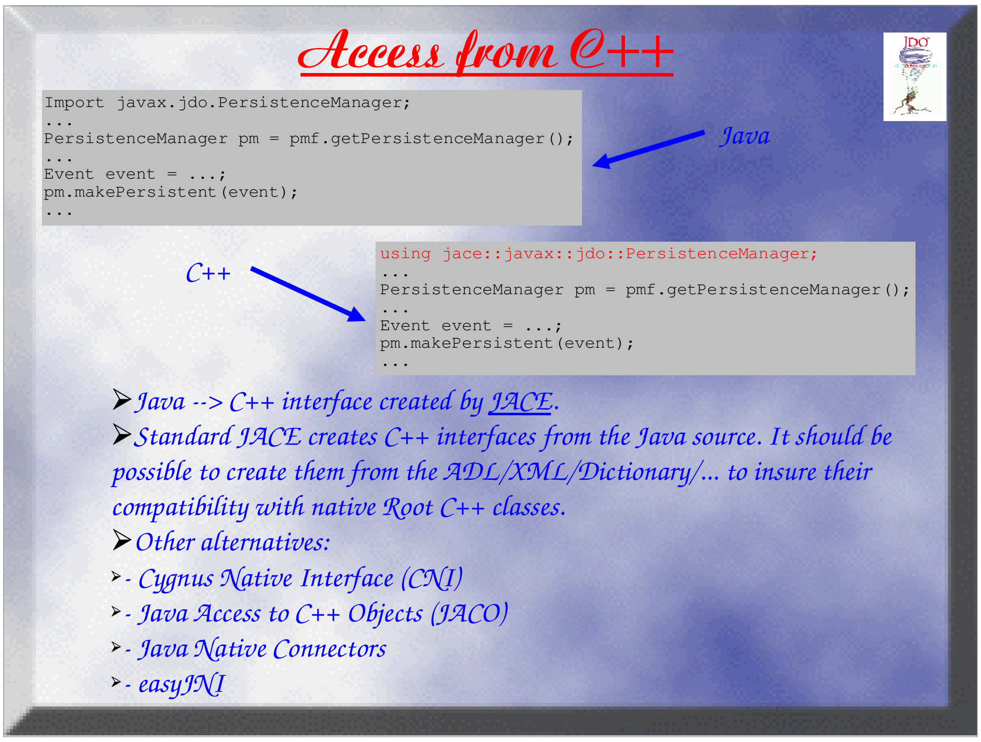
- *References within one JDO DB (Root file) are handled automaticaly.*
  - *OID is defined by Application to be unique, they do not contain any technology specific part.*
  - *OID is transparently assigned by PersistenceManager (using enhanced object).*
  - *Navigation Service (Grid, JNDI, simple table,...) maps OID into OLOC (Object Locator), which contains technology specific information.*
  - *Identity managed by Application (i.e. LCG identity) should be compatible with the native C++ OID.*
- *External References can be handled by PersistenceManagerFactory with connection to standard Navigational Service.*
  - *PersistenceManagerFactory creates proper PersistenceManager connected to JDO DB which containes required Object. The address (OLOC)of this JDO DB is obtained from the Navigational Service.*
  - *DynamicProxies can be used for transparent handling of the remote References at the language level. Those can also provide additional functionality (caching, lazy-loading, placement,...).*

•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.



RTAG Architecture - Misc

- Transient Dictionary is not needed thanks to Java Reflection. Persistent-Transient Dictionary (mapping) 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.



Access from C++

Java

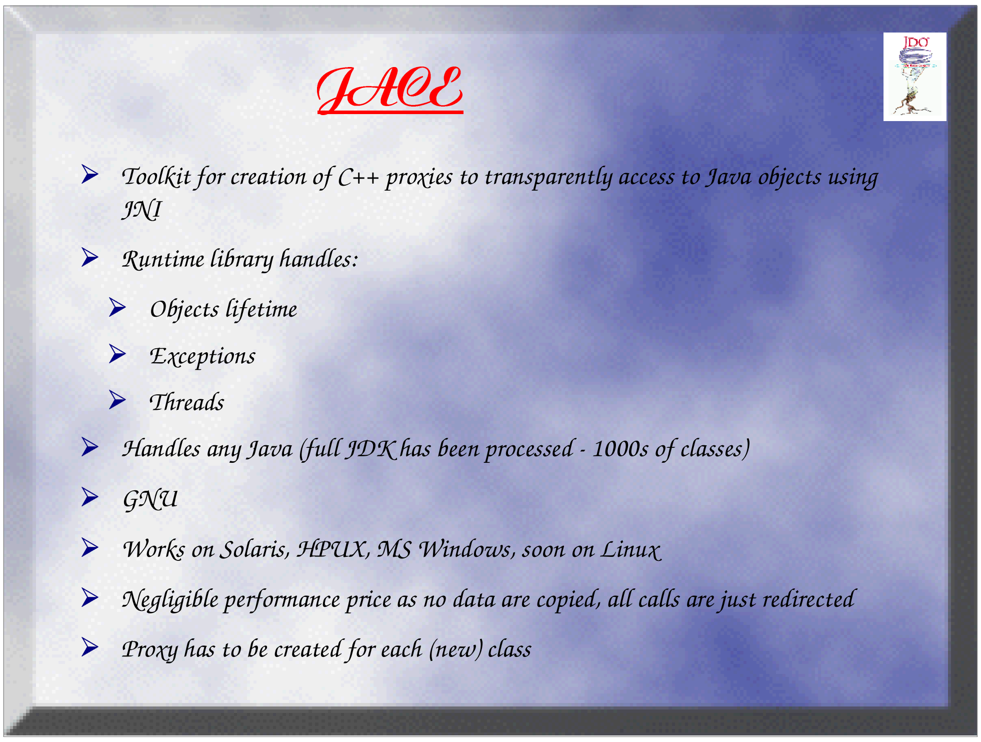
```
Import javax.jdo.PersistenceManager;
...
PersistenceManager pm = pmf.getPersistenceManager();
...
Event event = ...;
pm.makePersistent(event);
...
```

C++

```
using jace::javax::jdo::PersistenceManager;
...
PersistenceManager pm = pmf.getPersistenceManager();
...
Event event = ...;
pm.makePersistent(event);
...
```

- *Java --> C++ interface created by JACE.*
- *Standard JACE creates C++ interfaces from the Java source. It should be possible to create them from the ADL/XML/Dictionary/... to insure their compatibility with native Root C++ classes.*
- *Other alternatives:*
  - *- Cygnus Native Interface (CNI)*
  - *- Java Access to C++ Objects (JACO)*
  - *- Java Native Connectors*
  - *- easyJNI*

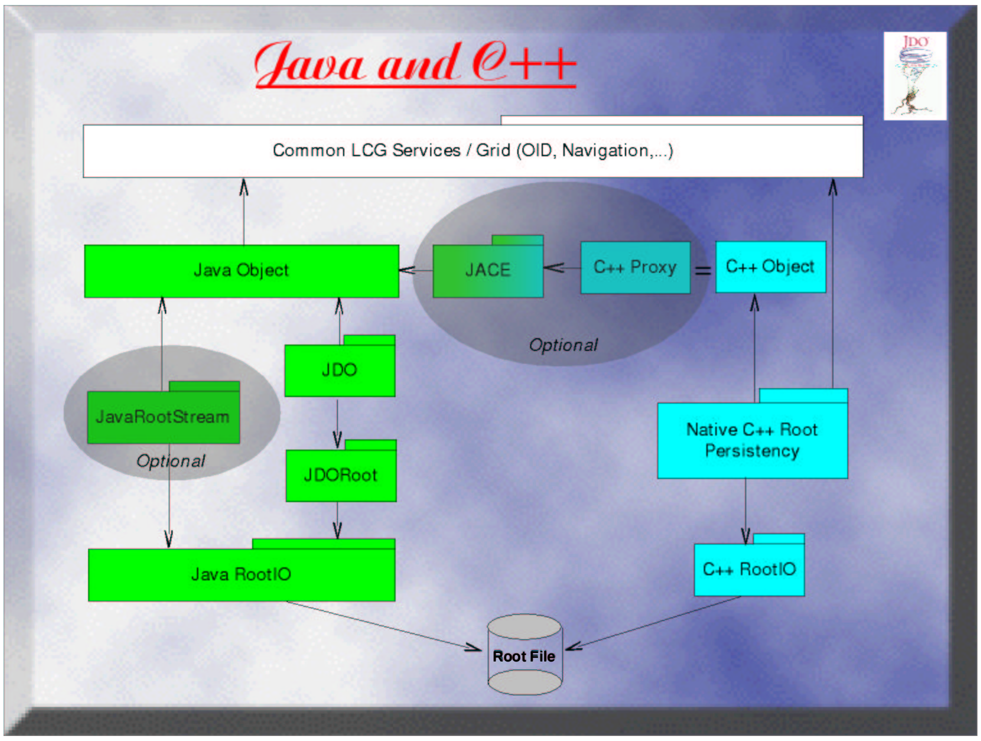
•JNI is not difficult, just tedious. It's important to get a tool which automates task of creation of interfaces. There are several candidates, JACE seems to be the best.



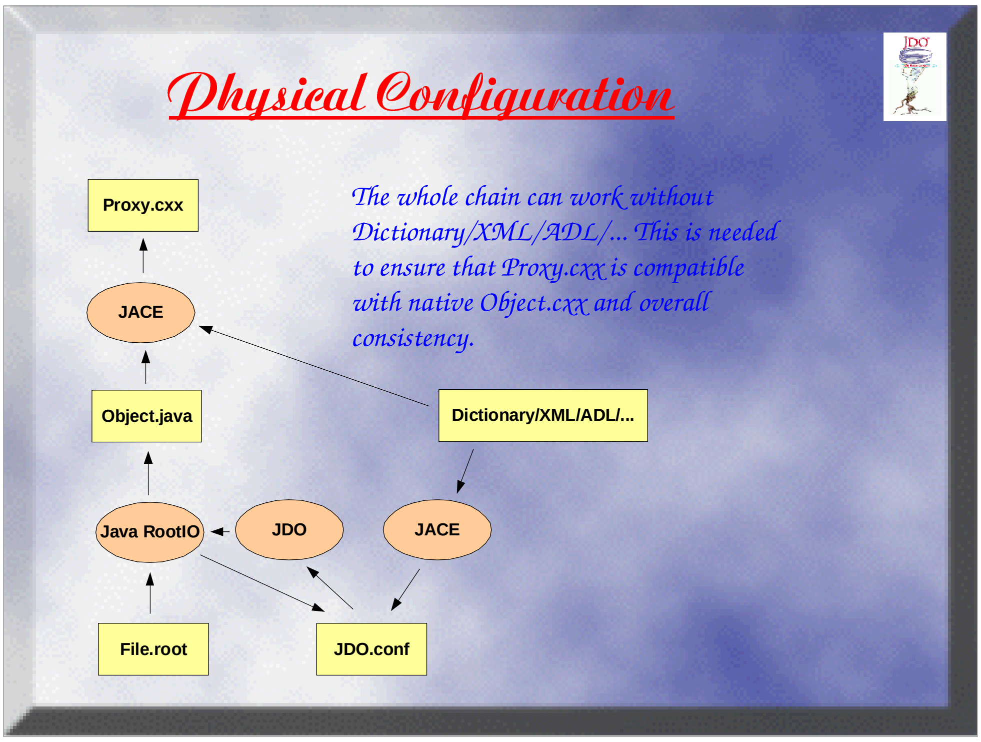
JACE

- Toolkit for creation of C++ proxies to transparently access to Java objects using JNI
- Runtime library handles:
  - Objects lifetime
  - Exceptions
  - Threads
- Handles any Java (full JDK has been processed - 1000s of classes)
- GNU
- Works on Solaris, HPUX, MS Windows, soon on Linux
- Negligible performance price as no data are copied, all calls are just redirected
- Proxy has to be created for each (new) class

•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.



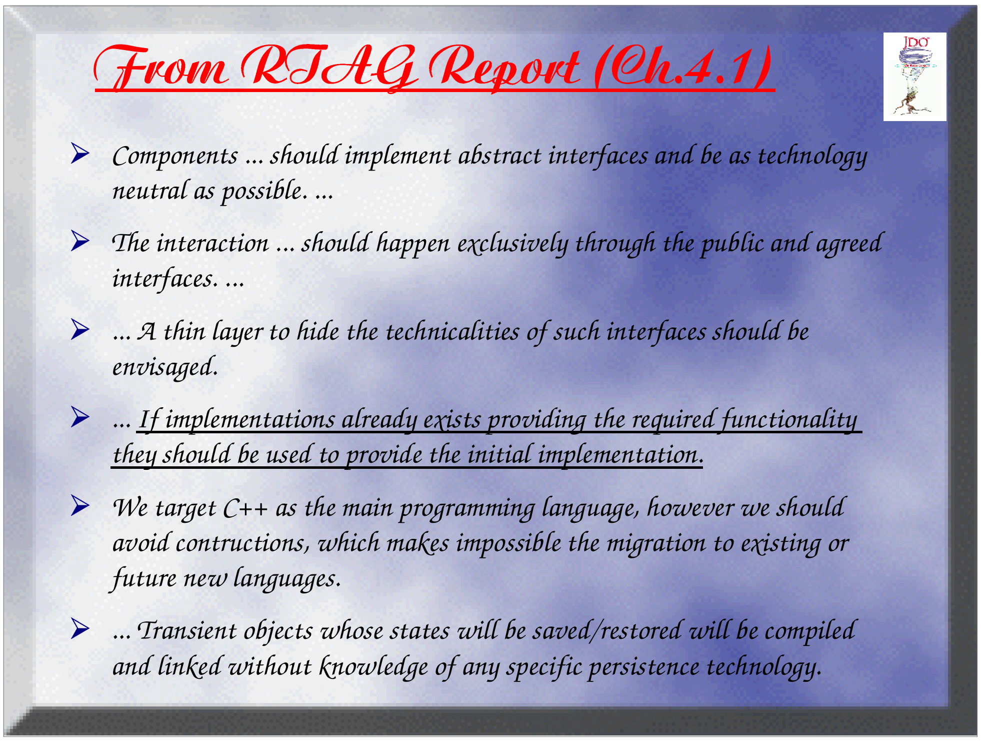
•Blue: C++.
•Green: Java.
•White: Language independent.



Physical Configuration

The whole chain can work without Dictionary/XML/ADL/... This is needed to ensure that Proxy.cxx is compatible with native Object.cxx and overall consistency.

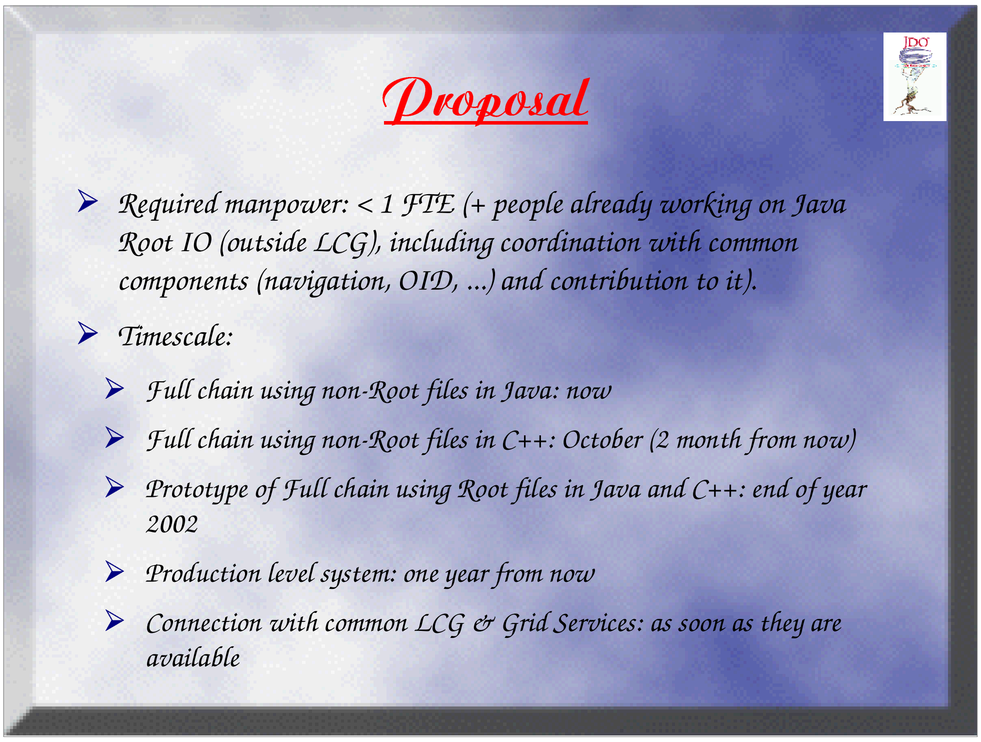
•Blue: C++.
•Green: Java.
•White: Language independent.



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.*

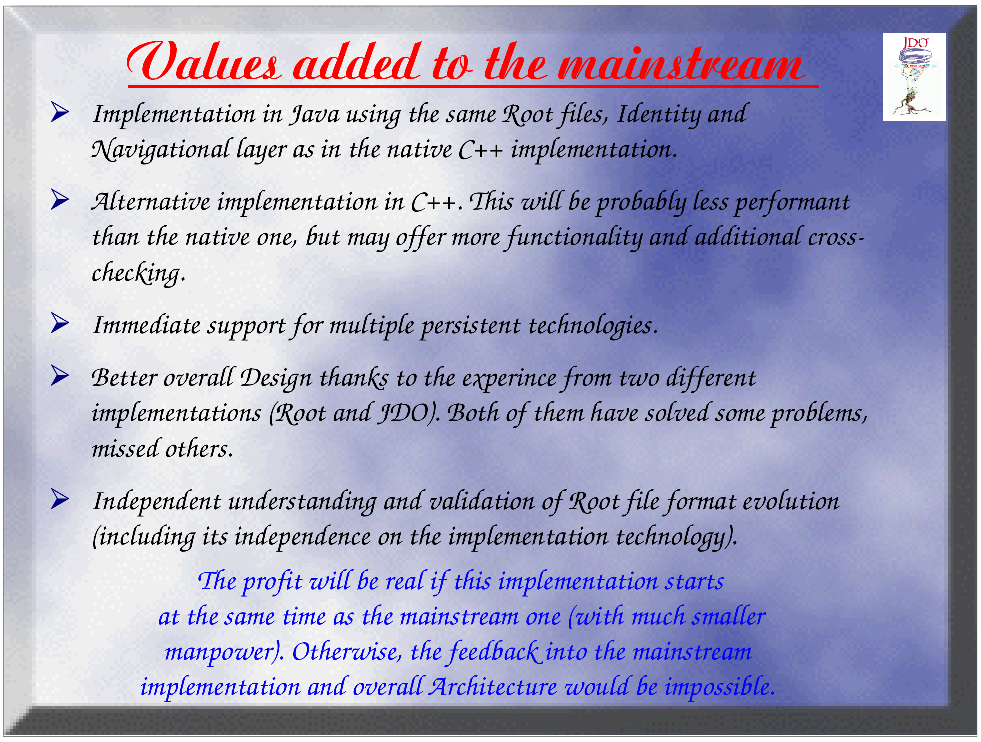
•Very well satisfied by JDO.



Proposal

- Required manpower: < 1 FTE (+ people already working on Java Root IO (outside LCG), including coordination with common components (navigation, OID, ...) and contribution to it).
- Timescale:
  - Full chain using non-Root files in Java: now
  - Full chain using non-Root files in C++: October (2 month from now)
  - Prototype of Full chain using Root files in Java and C++: end of year 2002
  - Production level system: one year from now
  - Connection with common LCG & Grid Services: as soon as they are available

•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.

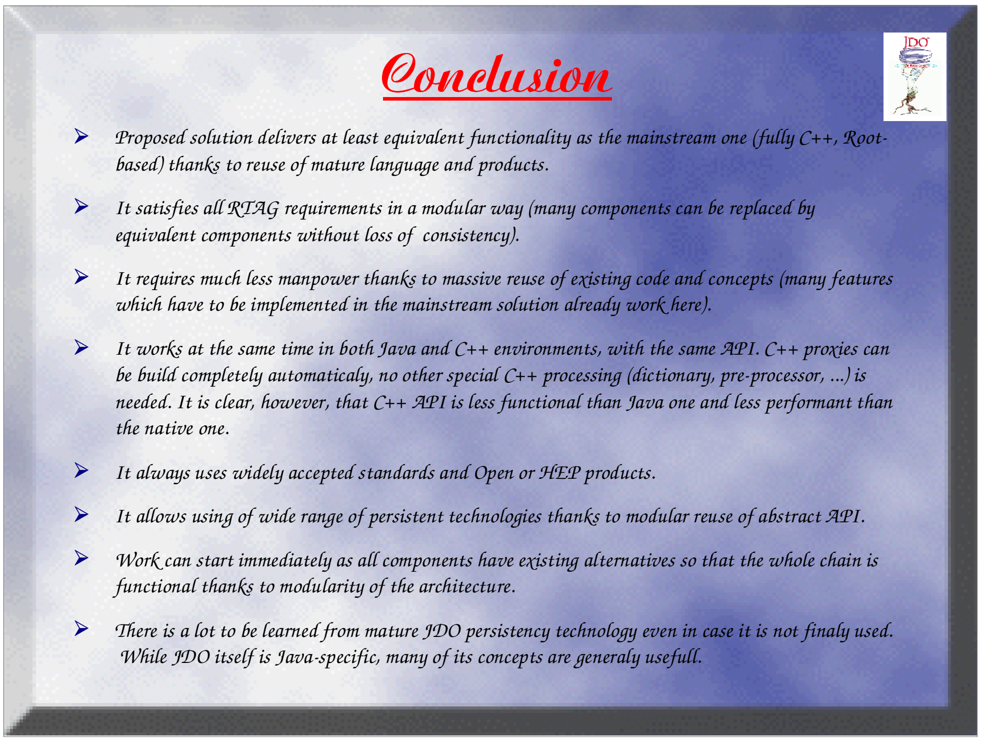


Values added to the mainstream

- Implementation in Java using the same Root files, Identity and Navigational layer as in the native C++ implementation.
- Alternative implementation in C++. This will be probably less performant than the native one, but may offer more functionality and additional cross-checking.
- Immediate support for multiple persistent technologies.
- Better overall Design thanks to the experince from two different implementations (Root and JDO). Both of them have solved some problems, missed others.
- Independent understanding and validation of Root file format evolution (including its independence on the implementation technology).

The profit will be real if this implementation starts at the same time as the mainstream one (with much smaller manpower). Otherwise, the feedback into the mainstream implementation and overall Architecture would be impossible.

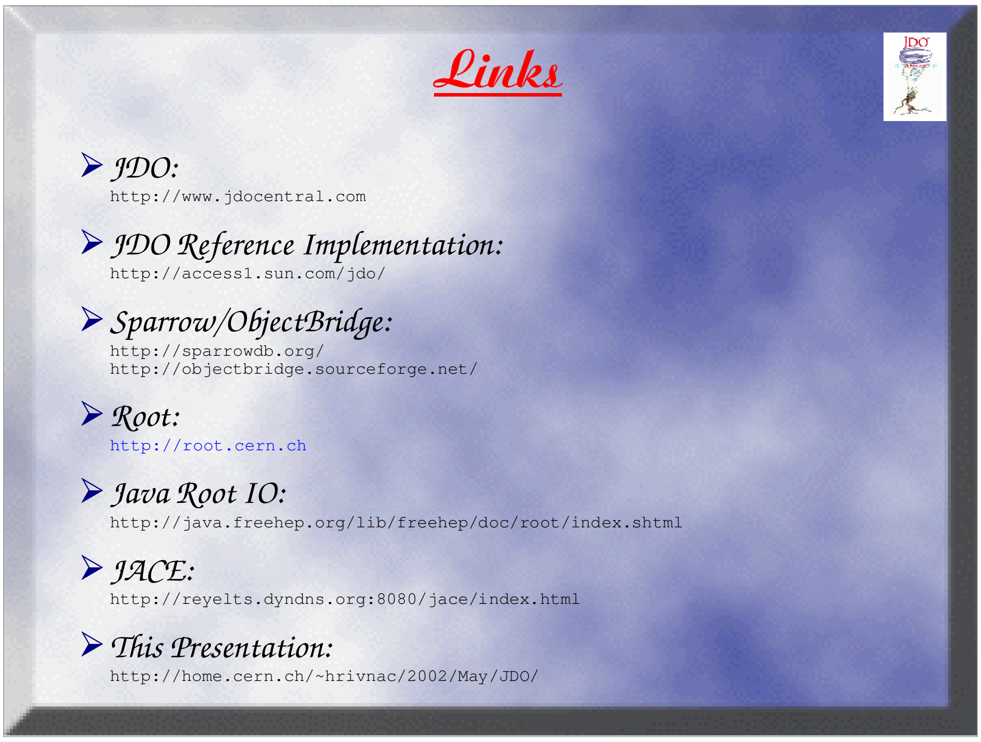
•The only guarantee of clean Interfaces is independent implementation.
•A lot of inconsistencies and bugs in the Root file format.



Conclusion

- Proposed solution delivers at least equivalent functionality as the mainstream one (fully C++, Root-based) thanks to reuse of mature language and products.
- It satisfies all 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 and less performant than the native 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.
- There is a lot to be learned from mature JDO persistency technology even in case it is not finaly used. While JDO itself is Java-specific, many of its concepts are generaly usefull.

•This project is possible thanks to heroic efford of Tony Johnson in decrypting native format of Root files.



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/