Sun

JavaOne

Developing an Object-Oriented Database for Embedded Systems on Java ME

Konstantin Knizhnik
Software Engineer, McObject

TS-5617

OODBMS for Java™ Platform, Micro Edition (Java ME Platform)

Implementing an Object-Oriented Database system for the Java ME application environment (based on Perst Lite experience)

Benefits, problems, and solutions

Agenda

- **Benefits of OODBMS for Embedded (Java ME platform) World**
    - Efficiency
    - Transparent persistence
- Aspects of embedded OODBMS architecture
    - Object cache
    - Transaction model
    - Memory allocation
    - Database-specific collections
- Specific of Java ME platform OODBMS implementation
    - Reflection replacement
    - Cache management
    - Storage layout

Object-Oriented Databases for Java Platform

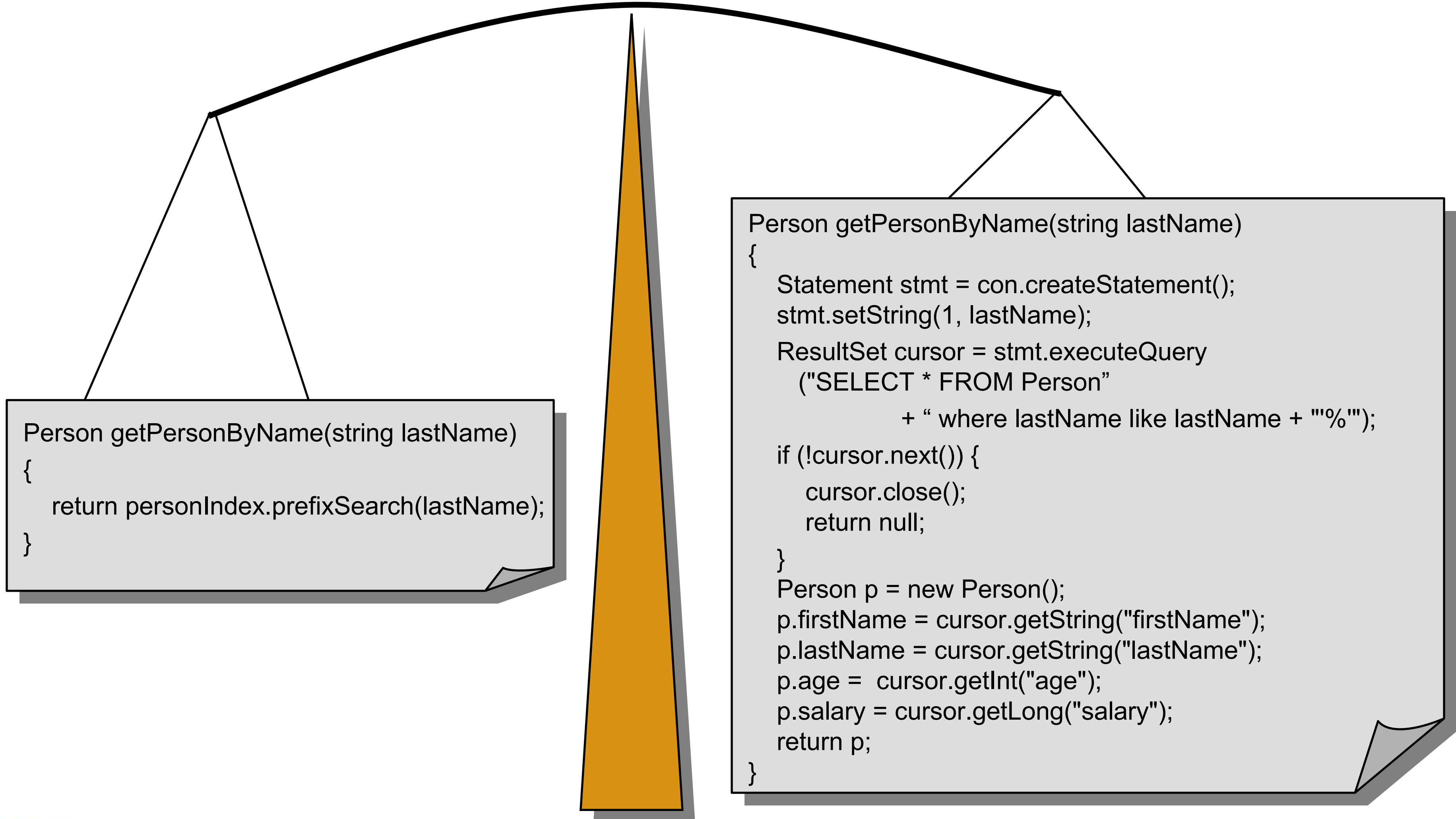
- Java ME platform: A runtime environment for devices
  - PDAs
  - TV set-top boxes
- Relational DBMS and client-server approach
- Embedded specifics
  - Local data processes
  - Simple queries
  - High speed and low footprint
- Object-Oriented database approach
  - Seamless interface with application
  - No records-to-objects pack/unpack overhead

JavaOne

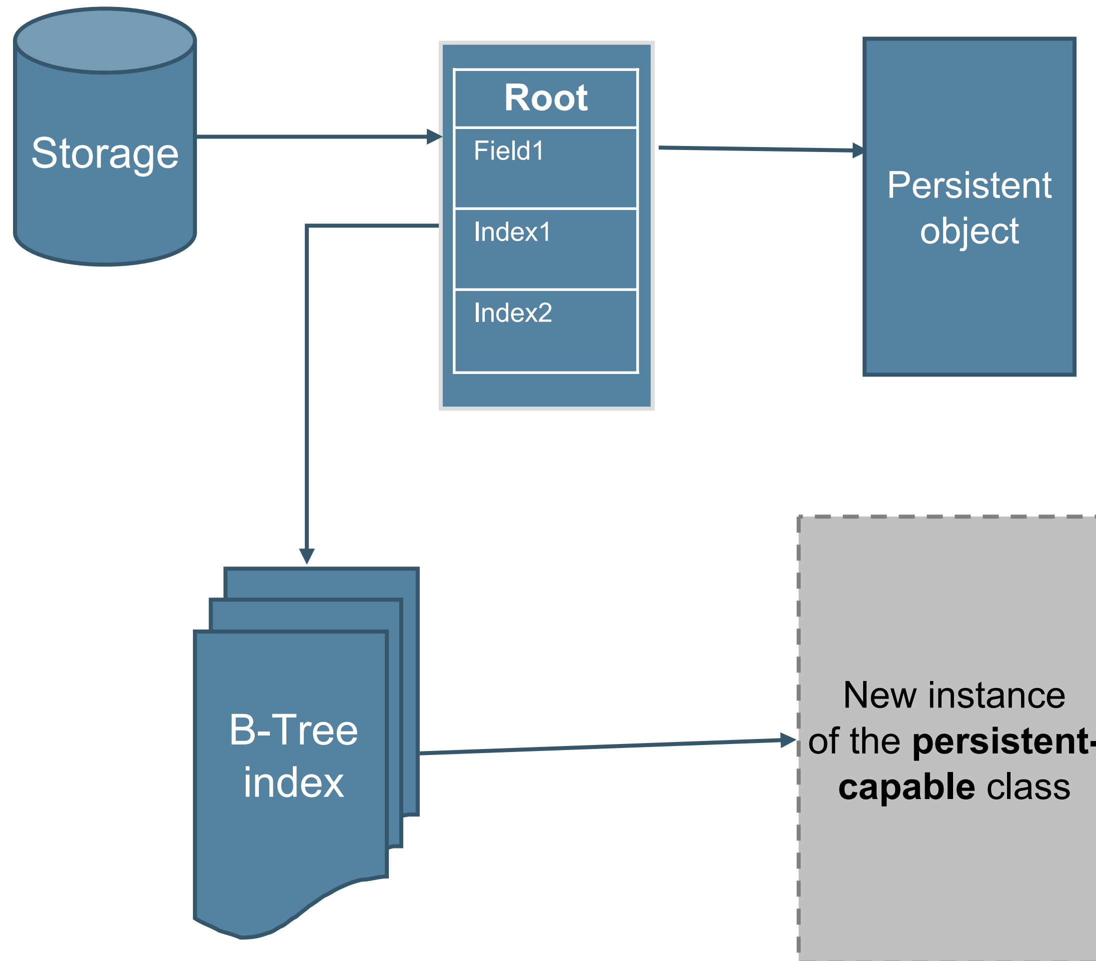
Efficiency Advantages of Object-Oriented Database Management Systems

- Single data model for the database and application
- No need to convert data from the “application’s representation” to the “database representation” and back
- No query processing overhead
- Efficient representation of complex data structures using object references
- Data encapsulation
- Polymorphic queries
- Use of a single programming language (as opposed to Java programming language and SQL)

Coding Efficiency: Object-Oriented Versus Relational Database Approaches



Achieving Persistence: Persistence by Reachability



Achieving Persistence by Reachability: Database Initialization

```
Storage db = StorageFactory.getInstance().createStorage();
db.open("test.dbs"); // open database
RootObject root = (RootObject)db.getRoot();
if (root == null) { // if database is not yet initialized
    root = new RootObject(db); // create a root object
    db.setRoot(root);          // and register it
}
...
db.close(); // close database
```

Achieving Persistence by Reachability: Code Sample

```
class Account {
     string id;
    long amount;

     public void deposit(long sum) {
           // update object field
           amount += sum;
          // mark object as modified
          modify();
    }
    Account (string id) {
           this.id = id;
    }
}
```

```
class Bank {
    FieldIndex<Account> accounts;
    void createAccount (string id) {
        // create new instance
        Account acc = new Account (id);
        // include it in index:
        accounts.add(acc); // acc is
        // made persistent implicitly
        return acc;
    }
}
```

Transparent Persistence

- Definitions
  - Transparent persistence: Data is accessed directly using Java programming language vs. a database sub-language (embedded SQL) or call interface (ODBC/JDBC)
- Java platform reflection support is a key to OOBDMSs' transparent persistence
  - Structural reflection: Inspection of object content at runtime
  - Lack of behavioral reflection: Not possible to control access to the objects
  - Not possible to implement completely transparent persistence without special tools
- Explicitly fetch/store persistent objects
  - Error prone
  - Lack of transparency: Eliminates the main advantage of OODBMS
  - Requires more effort from programmer

Transparent Persistence Benefits: Code Sample

Explicit control of object persistence

```
class Project {
      // use identifiers instead of references
      int managerID;
}
class Manager {
     int projectID;
     void assign(Project p) {
       // load object
       Manager mgr = db.get(p.managerID);
       // update object
       mgr.projectID = 0;
       // store modified object
       db.put(mgr);
      // get object ID
       this.projectID = db.getID(p);
     }
}
```

Transparent persistence: No difference between transient and persistent objects

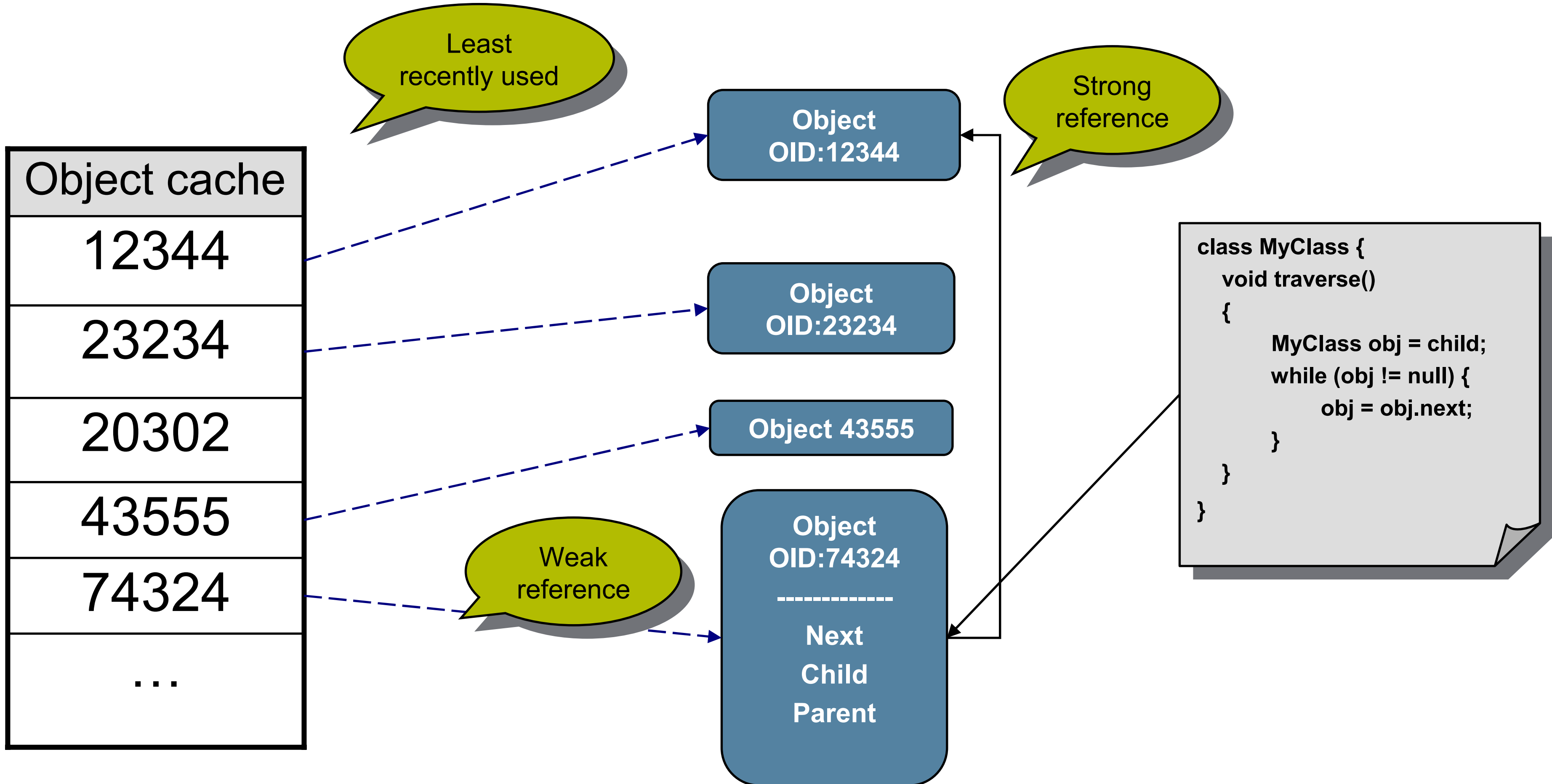
```
class Project {
  Manager manager; // use normal references
}
class Manager {
  Project  project;
   void assign(Project p) {
       // not necessary to load the object
      Manager mgr = p.manager;
       // update object
       mgr.project = null;
       // not necessary to explicit store the
      object
        project = p;
   }
}
```

Agenda

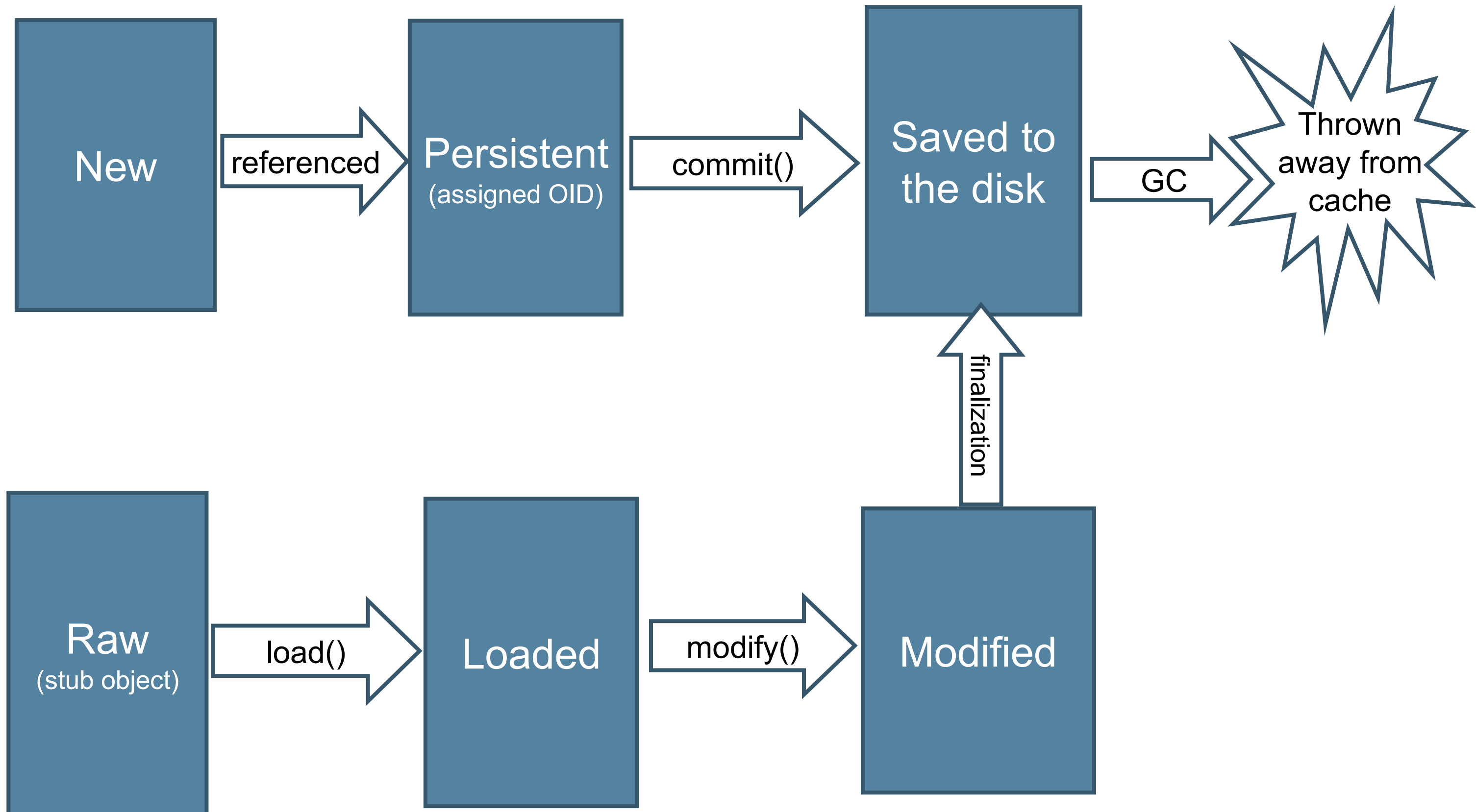
- Benefits of OODBMS for embedded world
  - Efficiency
  - Transparent persistence
- **Aspects of Embedded OODBMS architecture**
  - Object cache
  - Transaction model
  - Memory allocation
  - Database-specific collections
- Specific of Java ME platform OODBMS implementation
  - Reflection replacement
  - Cache management
  - Storage layout

Object Caching Keeps Frequently Used Objects in Memory

Goal: Avoiding Excessive Storage Access



Persistent Object State Transfer



Two OODBMS Transaction Models: Write Ahead Log and Shadow Objects

- Write ahead log
  - Based on writing all data changes into a log file before writing them into the database file; in case of fault, all committed transactions can be recovered from the log
- Shadow objects
  - The database creates copies of updated objects, and transactions modify these copies rather than the original objects; the database accesses objects (originals and copies) indirectly through the “object index”; transactions are committed by switching between the “active” and the “shadow” versions of this index

Write Ahead Log vs. Shadow Objects: Advantages and Disadvantages

Write ahead log	Shadow objects
Restricted by log size, system administrator interaction usually needed	No need for extra files
Always double amount of disk IO	If application is not using transactions, then overhead is minimal
Recovery requires reading and processing the whole log file	Fast recovery time—only need to switch current index
Easily facilitates multiple concurrent transactions	Concurrent transaction execution is problematic

Memory Allocation Strategies

- Memory allocation is a critical component of reaching top performance in object-oriented databases
- Memory allocators critically affect database performance through:
  - Allocation and de-allocation speed
  - Memory overhead
  - Fragmentation
- Memory allocators’ secondary impacts on performance
  - Locality of references
  - Protection from fault

Database-Specific Collections

- OODBMS offers classes for dedicated purposes, such as:
  - Range queries: “Select * from T where age between 20 and 40”
  - Optimize access to the disk: B-Tree
  - Lazy loading of collections’ members
  - Spatial indices: R-Tree
- Embedded collections: Reducing the number of objects
- Extends the limited set of standard collection classes available in Java ME platform and provides JDK 1.2 compatible abstractions

Example: Perst OODBMS Uses Its Range Queries Collection

```
// calculate average salary for employees which age
// belongs to the specified interval
void CalculateAverageSalary(Date min, Date max) {
   long totalSalary = 0;
   int nEmployees = 0;
   for (Employee e : index.iterator(new Key(min), new Key(max),
                    Index.ASCENT_ORDER))
   {
         totalSalary += e.salary;
         nEmplyees += 1;
   }
   return nEmployees != 0 ? totalSalary/nEmplyees : 0;
}
```

Agenda

- Benefits of OODBMS for embedded world
  - Efficiency
  - Transparent persistence
- Aspects of embedded OODBMS architecture
  - Object cache
  - Transaction model
  - Memory allocation
  - Database-specific collections
- **Specifics of Java ME platform OODBMS implementation**
  - Reflection replacement
  - Cache management
  - Storage layout

Java

JavaOne

Java ME Platform (CLDC 1.0 and 1.1) vs. Java Platform, Standard Edition (Java SE Platform) Supported Features

	CLDC 1.0	CLDC 1.1	J2SE™ Platform
Reflection			X
Weak references		X	X
Floating point		X	X

Engineering Around CLDC Limitations: Reflection

- Reflection is used to inspect the object format at runtime
- Reflection gives the database runtime “knowledge” of the storage format and access methods of the object

How OODBMSs Use Reflection to Pack and Unpack Objects

```
// List of field descriptors prepared by
// the OODBMS using reflection
FieldDescriptor[] flds = desc.fields;
// Loop through all fields in the class
for (int i = 0, n = flds.length; i < n; i++) {
    FieldDescriptor fd =flds[i];
    Field f = fd.field;

     switch(fd.type)  {
       case ClassDescriptor.tpByte:
            buf.extend(offs + 1);
            // get the value of a byte field
            buf.arr[offs++] = f.getByte(obj);
                    continue;
         case ClassDescriptor.tpInt:
             buf.extend(offs + 3);
                     // get the value of a  short field
                     Bytes.pack2(buf.arr, offs, f.getShort(obj));
     }
}
```

- Database engine iterates through all fields of the object, extracts their values and packs them into an internal database format
- Object is stored on disk as an array of bytes
- Reflection is slow because the runtime has to verify that object fields are accessed in the proper way

How Perst Lite Replaces Reflection

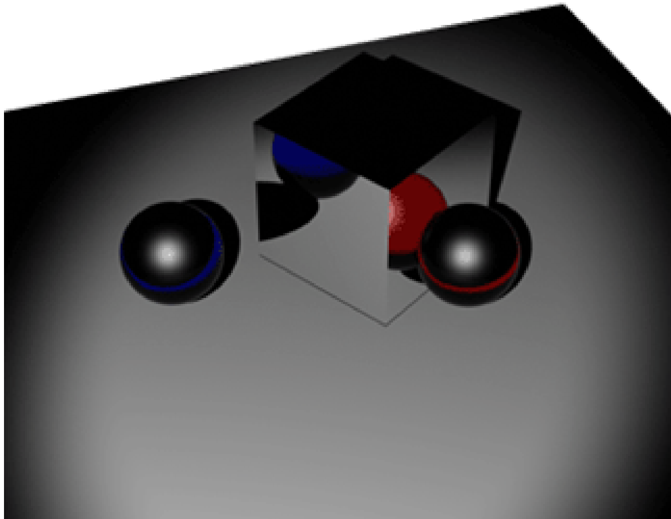
```
public class Manager extends Employee {
    public Project project;
    public int age;
    public String character;

    public void write (+ out) {
        super.write(out);
        out.writeObject(project);
        out.writeInt(age);
        out.writeString(character);
     }
}
```

```
final byte[] packObject(IPersistent obj) {
     // create stream for writing
     // serialized data
     PerstObjectOutputStream out =
                new PerstObjectOutputStream(obj);
     // write the object to the stream
     obj.write (out);
     // return byte array with the
     // serialized data
     return out.toArray();
}
```

- Pack/unpack routines are automatically generated by the preprocessor
- Less overhead: no loops and switches
- Need for default constructor is eliminated

Engineering Around CLDC Limitations: Object Cache



- The “object” is the fundamental concept in OODBMS and applications usually access a large number of objects
- An OODBMSs’ object cache keeps frequently used objects in memory, to avoid excessive access to non-volatile device storage

Overview of Caching Policies

Strong	Weak	Soft	Pinned	LRU (least recently used)
Object cache keeps strong references to all objects This cache "pins" all objects in memory and is used for in-memory databases	Object cache is based on the **java.lang.WeakReference** class Un-referenced objects in the cache live until the first garbage collection (GC)	Object cache is based on the **java.lang.SoftReference** class GC will do its best to keep objects in the cache as long as possible	Variation of the weak cache. Objects are pinned in memory until the transaction commit Eliminates disk I/O until the commit and avoids redundant I/O	Implemented upon weak cache, pins the most frequently used objects in memory using the LRU discipline

OODBMS Caching in the Java ME Application Environment

- An OODBMS's object cache keeps frequently used objects in memory, to avoid excessive access to non-volatile device storage;
lack of weak references requires using strong references and cache clean-up upon transaction commit
- It is still possible to use the LRU discipline to keep the most frequently used objects in cache

JavaOne

Example of Java ME Platform OODBMS Using Object Cache for Object Lookup

```
final IPersistent lookupObject(int oid, Class cls) {
  IPersistent obj = objectCache.get(oid); // try to find object in cache
  if (obj == null || obj.isRaw()) {
    obj = loadStub(oid, obj, cls); // object is not in the cache or is raw
  }
  return obj;
}
final IPersistent loadStub(int oid, IPersistent obj, Class cls)
{  …
    if (obj == null) {
       obj = (IPersistent)desc.newInstance(); // create new instance
       objectCache.put(oid, obj); // put created instance in the cache
    }
    return obj;
}
```

Perst Lite’s Storage Layer for Java ME Platform

Database Storage

- Use page pool to optimize access to the underlying storage
- Abstract from the underlying implementation

Record Management System
Package: javax.microedition.rms

- One virtual file on top of several physical files
- Database page is stored as RMS record

Java™ Specification Request (JSR) 75
Package: javax.microedition.io.file

- Access to any file and device
- Available for PDA and smartphones

Conclusion

- OODBMSs in the embedded world must operate in a restricted environment; this demands efficient and compact algorithms
- Features traditionally used in Java platform OODBMSs such as reflection and weak object cache have to be replaced with lightweight alternatives
- In order to be truly useful, an object-oriented embedded database for Java ME platform must not only adhere to CLDC specifications, but also accommodate development in specific environments, such as Blackberry
- Complexity of Java ME platform storage management argues in favor of integrating open source or commercial database system rather than self-developed solution
- Perst.Lite (www.mcobject.com/perst) is an Open Source, truly object-oriented DBMS for Java ME platform that is built on the principles discussed in this presentation

For More Information

- Visit McObject website
  http://www.mcobject.com/perst/
- Sun resources
  - http://java.sun.com/javame/index.jsp
  - http://java.sun.com/javame/technology/index.jsp
  - http://java.sun.com/products/midp/
  - http://java.sun.com/products/sjwtoolkit/
  - https://phoneme.dev.java.net/content/index_feature.html
- Vendor-specific Blackberry development tools
  http://na.blackberry.com/eng/developers/

Q&A

Konstantin Knizhnik, McObject
Andrei Gorine, McObject

JavaOne

Developing an Object-Oriented Database for Embedded Systems on Java ME

Konstantin Knizhnik
Software Engineer, McObject

TS-5617