ETH Zürich
Chair of Software Engineering

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Java: Persistence

Outline

- Java Serialization
- Connecting to a Relational Database Management System (RDBMS) with Java Database Connectivity API (JDBC)
- Object-Relational Mappers (ORM) and Data Mappers
- Object-Oriented Data Base Management Systems (OODBMS)

Java Serialization

Java Binary Serialization

- Objects are stored in a file together with their object graph and can be retrieved from compatible class versions
- Your custom objects need to implement either **`Serializable`** or **`Externalizable`**
- **`static`** and **`transient`** fields will not be serialized

Writing and reading objects

- Write objects to a **`FileOutputStream`** using method **`writeObject`** in **`ObjectOutputStream`**
- Read objects from a **`FileInputStream`** using method **`readObject`** in **`ObjectInputStream`**

- Write primitive data types using methods in interface **`DataOutput`**
- Read primitive data types using methods in interface **`DataInput`**

- Write bytes and arrays of bytes using **`OutputStream`**
- Read bytes and arrays of bytes using **`InputStream`**

Java Binary Serialization example

```
class ClassA implements Serializable
{
   private int field1;
   private ClassB field2;
   private transient String field3;
   ...
}
```

Sample serialization

```
class ClientClass
{
   public void serialize(Serializable target)
   { //exception handling omitted
     FileOutputStream os = new
     FileOutputStream("fileName");
     ObjectOutput oo = new
     ObjectOutputStream(os);
     oo.writeObject(target);
     oo.close();
   }}
```

Sample deserialization

```
class ClientClass
{
   public Object deserialize()
   {//exception handling omitted
     FileInputStream is = new
     FileInputStream("fileName");
     ObjectInputStream oi = new
     ObjectInputStream(is);
     ClassA obj = (ClassA)oi.readObject();
     oi.close();
     return obj;
   }}
```

Serializable vs Externalizable

- The serialization mechanism triggered by implementing **`Serializable`** uses reflection. In early versions of Java this created performance issues on large objects
- For efficiency you can use e.g. JBoss serialization library: http://www.jboss.org/serialization
- You can fully customize the serialization mechanism by implementing the **`Externalizable`** interface
- **`Externalizable`** delegates to the class complete control over the storable external format by asking you to implement **`writeExternal`** and **`readExternal`**

A deceivingly simple choice

- Implementing the `Serializable` interface is a strong design choice, because it makes the class instances persistent forever
- The class implementation (e.g. the `private` fields) is now part of the default serialized form, that is, part of the class API
- Descendants and supplier classes (e.g. `ClassB`) need to be `Serializable` as well

The serial version UID

- Every `Serializable` class has a unique identification number associated with it, the serial version UID

```
private static final long serialVersionUID
```

- This requirement is in place to prevent accidentally evolved classes

- If we don't provide this constant field and the associated value, the compiler warns us. This is a warning it is not safe to ignore

Handling the serial Version UID value

Choice 1: accept the default generated UID

- Just choose the option of generating one when using Eclipse or simply ignore the compiler warning
- Most of the changes in the class structure will now trigger a mismatch (`InvalidClassException`)

Choice 2: provide a custom value

- Adding an attribute will now cause no exception
- The added attribute will be initialized to its default value

Providing a custom serialized form

- Implement `Serializable` providing your own version UID
- Replace the default serialization mechanism using the following callback methods in your class:

```
private void writeObject(ObjectOutputStream s)

private void readObject(ObjectInputStream s)
```

- Invoke the default serialization mechanism from their bodies with

`s.defaultWriteObject()` and
`s.defaultReadObject()`

Deserialization and schema evolution

Deserialization is an extra-linguistic mechanism to create objects

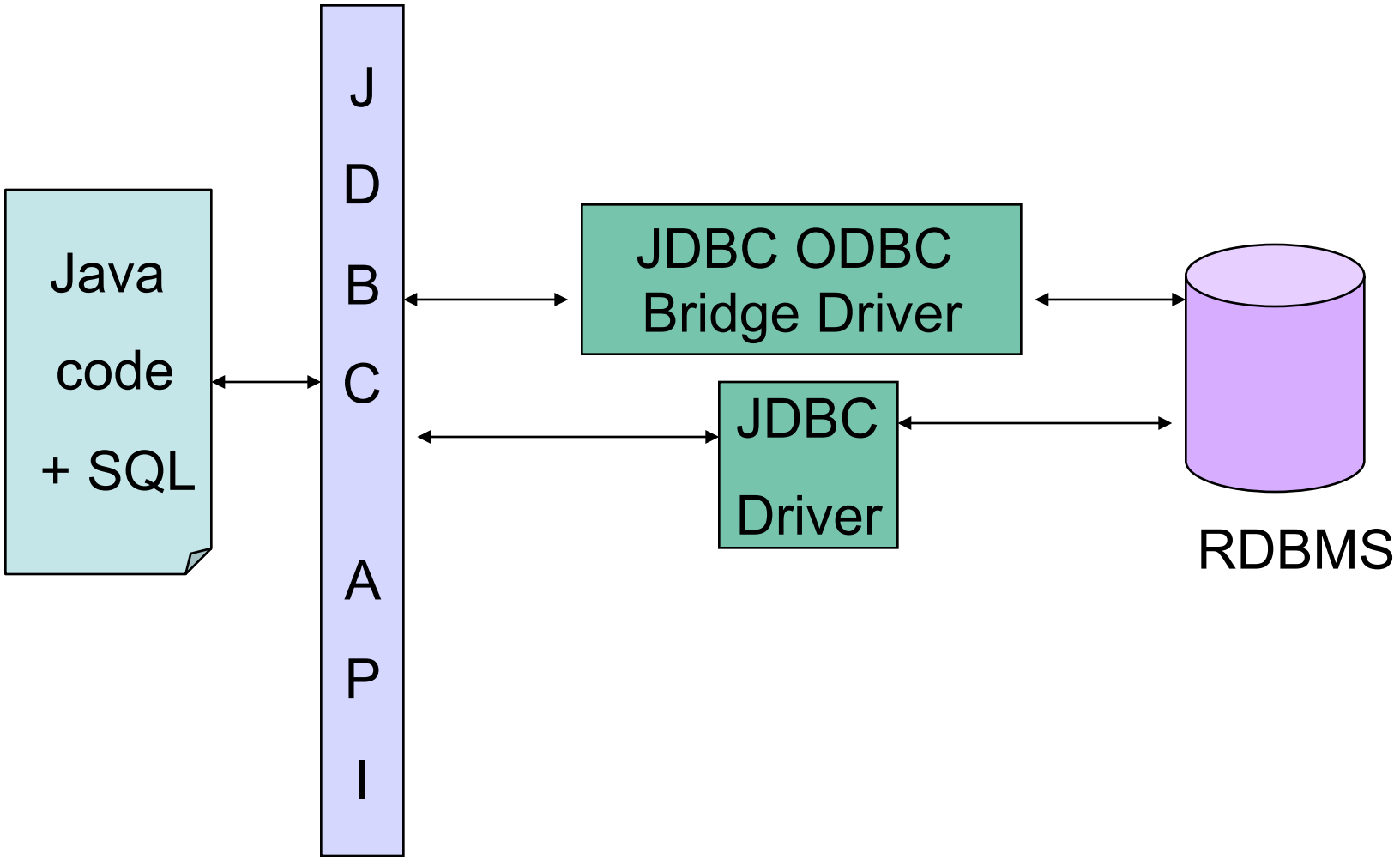
- Object have to be reconstructed without using constructors
- Possible security issues
- Possible correctness issues (class invariant violations)

Beware of “transparent schema evolution” claims

- Providing a default value to a newly added attribute when reading an object of an old version can be wrong
- Use `readObject` to re-establish the class invariant

Connecting to a Relational Database Management System (RDBMS) with Java Data Base Connectivity API (JDBC)

Java Data Base Connectivity API



What is a Relational DB in one slide

A collection of relations (tables)

Each relation is defined as a set of tuples (rows) having the same attributes (columns)

Each tuple (row) represents an object together with the related information about it

Each attribute (column) references data in the same domain

Data are accessed in a declarative way, specifying queries

A sample table: CUSTOMERS

Customer_id	Customer_ name	Customer_ country	Customer_data
123	Reto	Switzerland	100000
132	Cecilia	Italy	300000
213	Hauke	Germany	200000
231	Nadia	Russia	400000
312	Yu	China	500000
321	Viswanathan	India	600000

Step 1: loading a driver

- `DriverManager` is the basic class for managing a set of JDBC drivers
- The following loads and creates an instance of a driver and registers it with the `DriverManager`

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")
```

- How can all this happen with just one instruction?

Step 2: getting a connection

- `java.sql.DriverManager` provides a connection to the DB
- The connection object will communicate with the DB

```
Connection con = DriverManager.getConnection
  ("jdbc:odbc:MyDSN", "aLogin", "aPassword");
```

- Here the code just knows about the DSN (Data Set Name) which is in turn mapped to the real database name outside the application
  - In Windows via Control Panel (ODBC Data Source)
  - In Mac OS X via ODBC administrator
  - In Linux updating "*.odbc.ini*"

Other ways to get a connection

- The example just seen used ODBC and DSN

```
Connection con = DriverManager.getConnection
  (“jdbc:odbc:MyDSN”, “aLogin”, ”aPassword”);
```

- We can also use ODBC without DSN

```
Connection con = DriverManager.getConnection
  (“jdbc:odbc:Driver=...”);
```

- Or a pure JDBC driver

```
Connection con = DriverManager.getConnection
  (“jdbc:aConnectionString;Login;Password”);
```

Connecting via a *DataSource* object

- The most common option in J2EE (Java 2 Enterprise Edition) applications
- A factory for connections to the physical data source
- Handles connection life-cycle through connection pooling
- An object that implements the *DataSource* interface will typically be registered with a naming service based on the Java Naming and Directory Interface (JNDI) API
- Implemented by each driver vendor

Step 3: statements, queries, updates

- A `java.sql.Statement` object encapsulates an SQL statement

```
Statement stmt = con.createStatement();
```

- SQL update, insert or delete use the same command

```
stmt.executeUpdate("INSERT INTO CUSTOMERS
  VALUES ("0123","Scott","Canada", 700000)");
```

- Simple SQL query

```
stmt.executeQuery("SELECT * FROM CUSTOMERS;");
```

CUSTOMERS table after insertion

Customer_id	Customer_ name	Customer_c ountry	Customer_data
1230	Reto	Switzerland	100000
1320	Cecilia	Italy	300000
2130	Hauke	Germany	200000
2310	Nadia	Russia	400000
3120	Yu	China	500000
3210	Viswanathan	India	600000
0123	Scott	Canada	700000

Step 4: handling a query result set

- `executeQuery()` returns a `java.sql.ResultSet` object

```
ResultSet rs = stmt.executeQuery("SELECT…");
```

- The `ResultSet` object will be used to iterate through the result
  - Method `next()` moves the cursor to the next row, and returns false when there are no more rows
  - There are *getter* methods (`getBoolean()`, `getLong()`, etc.) for retrieving column values from the current row

Step 5: cleaning up

- Even if using a *DataSource* connection pooling, remember to explicitly destroy the connection objects after finished

```
...
// Also closes the result set
stmt.close();
// Very important!!!
con.close();
...
```

- How can you be sure that a connection is always closed?

Step 5: cleaning up in Java 7

```
try(Connection c = getConnection(…)) {
  try (Statement s = c.prepareStatement(…)){
        // work with PreparedStatement
        c.commit;
  } catch (SQLException e){
        // handle exception
        c.rollback();
        // maybe re-throw, or
        //wrap exception and then re-throw
        ...
  }}
```

Possible issues with sql statements

Across different DBMS

- Strings quoting may be different
- Code gets bloated with ugly string concatenations

```
String query = "Select * from Customers where
  " + " customer_id = \"" + id + "\"" +
  " and customer_country = \"" + country +
  "\"“ + ” and customer_data > “ + data + ”;“;
```

Characters that create conflicts are escaped with a backslash (e.g.: \“)

Prepared statements

Encapsulated, pre-compiled queries

- More readable, more portable
- Favor query optimization

```
String SQL = "select * from Customers where
  customer_id = ? and customer_country = ? and
  customer_data > ?";
PreparedStatement pstmt =
  con.prepareStatement(SQL);
pstmt.setString(1,id);
pstmt.setString(2,country);
pstmt.setInt(3,data);
pstmt.executeQuery();
```

Issues with JDBC

- JDBC is a good, but quite low level API
- It requires effort on the developer's side to write and maintain glue code
  - Loading and registering drivers
  - Handling connections
  - Handling exceptions (JDBC API provides only a few)
  - Handling Create, Read, Update, and Delete (CRUD) operations in a standard way
- It's better to use a good framework to handle all this

Spring JDBC

Spring is the leading open source framework for J2EE applications. Supports JDBC by providing:

- Configurable classes implementing **`javax.sql.DataSource`**
- Many useful runtime exceptions, mapped to db-specific errors by an **`SQLExceptionTranslator`**
- Classes like **`JDBCTemplate`** to handle the core workflow
- Developers only need to implement callback interfaces to handle the mapping

Object-Relational Mappers (ORM)
and
Data Mappers

Object-relational impedance mismatch

- Object oriented applications and relational databases implement different mathematical models
- A mapping layer is therefore necessary to represent objects as tables and vice versa
- Mapping objects to tables may be difficult and error prone
- From here stems the idea of ORM's and data mappers

ORM: Object Relational Mappers

- Mapping is handled in configuration files, or automatic
- Use SQL or ad hoc object query languages
- Have different, pluggable caching strategies
- Automatically detect updated objects and persist them
- Examples: Hibernate (Java and .NET), JPA (Java Persistence API)

ORM: when they are a good choice

- Typical load-edit-store workflows
- In need of occasional queries over big sets (SQL Unions) with single updates or deletes
- Read-mostly scenarios (web-like)
- Natural mapping between objects and tables
  - For example when there are flat object structures

ORM: when they are questionable

- Lots of set accesses (SQL Unions)
- Lots of aggregate functions
  - AVG, COUNT, MAX, MIN, SUM
- Lots of batch updates on multiple lines
- In need of specific SQL optimization
  - In presence of deep and/or recursive object structures

Data Mappers

- Think about them as lightweight ORM's
- Provide support for caching query results
- SQL and mappings are externalized in configuration files
- No automatic conversion of values to/from db
- No automatic updates of modified objects
- Example: MyBatis

Object-Oriented Data Base Management Systems (OODBMS)

OODBMS

Offer more functionalities than object serialization

- ACID Transactions (see next slide)
- Queries

Will probably replace serialization in the long run

Examples: Versant db4objects, McObject Perst

- Open source, available for both Java and .NET
- Small memory footprint, good to embed
- Easy to use

Database transactions ACID properties

Atomicity

- Either all the tasks of a transaction are executed correctly (commit), or none (rollback)

Consistency

- The database remains in a consistent state before and after the transaction

Isolation

- Data involved in a transaction are isolated from outside
  - Data in an intermediate state cannot be seen from outside the transaction

Durability

- Once committed, a transaction will survive also system failure

Storing an object with db4o

Suppose `Deck` is a "pure" business class

- Not "polluted" with persistence code
- Does not implement or extend anything

```
public void store(Deck aDeck)
{
  ObjectContainer db =
  Db4o.openFile("myDb.yap");
  db.store(aDeck);
  db.commit();
  db.close();
}
```

Querying for objects

The most interesting approach uses native queries

- Use Java to query for objects
- Different constraints on objects are possible
- Inherently type safe (compile-time checks)

Db4o native query example

```
...

public void testQuery()
{
  List<Student> result = objContainer.query(
  new Predicate<Student>()  {
   public boolean match(Student stu)   {
         return stu.getTopMark() == 6;
   }
  });
...
```