Java AntiPatterns

- Common mistakes
- Best Practices
- Coding standards

Govind Krishna Mekala

What we won’t cover

- J2ee/EJB antipatterns
- Design Patterns

What we will cover

Interactive session with lots of examples on:

- Core Java mistakes and solutions.
- Best Practices for Java and JDBC.
- Why coding standards are so important?

Object Creation

- Static factory methods are more appealing than constructors.

```
Public static Boolean valueOf(boolean b) {
    return (b?Boolean.TRUE:Bolean.FALSE);
}
```

- Use private constructor when defining a class for grouping static fields and methods.

- Avoid creating duplicate objects:

  String s = new String("No");
- Eliminate obsolete reference by nulling out to avoid memory leaks due to unitentional object reference. (Specially in collections and Maps).
- Nothing time critical should ever be done by a finalizer. Never depend on Finalizer and we can't time GC. Use finalizer as safe net to terminate noncritical native resources without forgetting super.finalize.

- Providing a good “toString” implementation makes your class much more pleasant to use.

```
System.out.println(“Failed to connect: “ +
    phoneNumber);
```

- Always override hashCode method when you override equals. Failure to do so will result in violation of the general contract for Object.hashCode which will prevent your class from functioning properly with conjunction with all hash-based collections, including HashMap, HashSet and HashTable.

Composition vs Inheritance

- Let's discuss this first.....
- Never extend a class unless documented to do so. Always document if you write a class for extending. Reason: Subclasses will break when super class changes. Unintentional trap for super class just because some ignorant programmers written several subclasses.
- Must read link: http://www.artima.com/designtechniques/compoinh.html

Prefer Interface over Abstract

- Multiple inheritance problem.
- Interface enables safe, power funtionality enhancement without using hierarchical type framework.
- You can combine the virtue of interface and abstract classes by proving as abstract skeleton implementation class to go with each nontrivial interface you export. E.g AbstractCollection, AbstractSet,AbstractList..
- Comparative analysis (Not official):

http://mindprod.com/jgloss/interfacevsabstract.html

- Use overloading carefully.
- Return zero-length arrays instead of nulls.
- Minimize the Use of Finalizers
- Avoid float and double if exact answers are required. Use BigDecimal, int or long for monetary calculations.

```
//0.61000000000001
    System.out.println(1.03 – 0.42);
//0.09999999999999995
    System.out.println(1.00 - .10);
```

Money matters in financial calculations.htm

- Avoid string concatenation for more than few strings. Use stringBuffer instead as string is immutable.
- Refer to objects by their interfaces if appropriate interface exists. (*flexibility)*.

```
List myList = new vector();
```

Exception Handling common mistakes

- **Empty Catch Blocks**
- **Loss of Stack Information.** Use exception chaining (or nesting) supported in JDK 1.4
- **Incomprehensible and Incomplete Exception Logs.**
- Because, a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. So, the subclass would never be reached if it come after its superclass. For example, **ArithmeticException** is a subclass of **Exception**

Enhanced Exception Handling Support in Java 1.4

- **Exception Chaining:** In Java version 1.4 support for it has therefore been built into the uppermost exception base class:

  Throwable(String message, Throwable cause)

  Throwable(Throwable cause)

- **Assertions:** They make it possible to improve software quality by abundant use of runtime checks for internal preconditions, postconditions and invariants similar to C assertion macros.

Thread Safety

- Synchronize access to shared mutable data. In a multi-threaded environment, *any* mutable data visible to more than one thread must be referenced within a synchronized block. This includes all primitive data. All get and set methods for shared data which can change must be synchronized.
- It is a misconception that all primitives except long and double do not need synchronized access.

Thread Safety cont..

- Avoid excessive synchronization. Excessive synchronization might causes deadlock.
  - Do as little as possible inside synchronized region. Obtain the lock, examine the shared data, transform the data as necessary, and drop the lock.
  - If a class could be used both in circumstances requiring synchronization and circumstances where synchronization not required, provide both the variants through wrapper class or subclass with synchronization.
  - To avoid deadlock and data corruption, never call an alien method from within synchronized region.

Thread Safety cont..

- Never invoke wait outside a loop and prefer notifyAll in preference to notify.
- Don’t depend on thread scheduler and thread priorities. They are not portable across VM and OS.
- Avoid thread groups. Thread groups were introduced for isolating applets for security which didn’t fulfill the promise. The APIs to get list of active threads and subgroups are buggy. Thread groups are obsolete.

Thread Safety cont..

- Singleton pitfalls. To decide whether to create a singleton class, first ask these questions:
  - 1. Does there need to one global entry point to this class?
  - 2. Should there be fixed number of instance in VM?
  - Never use singleton class as global variables as they might become non singleton.
- Read link: http://www.artima.com/designtechniques/threadsafety.html

JDBC

- Use connection pooling.
- Choose appropriate isolation levels.

A transactional system has to be able to deal with concurrency issues :

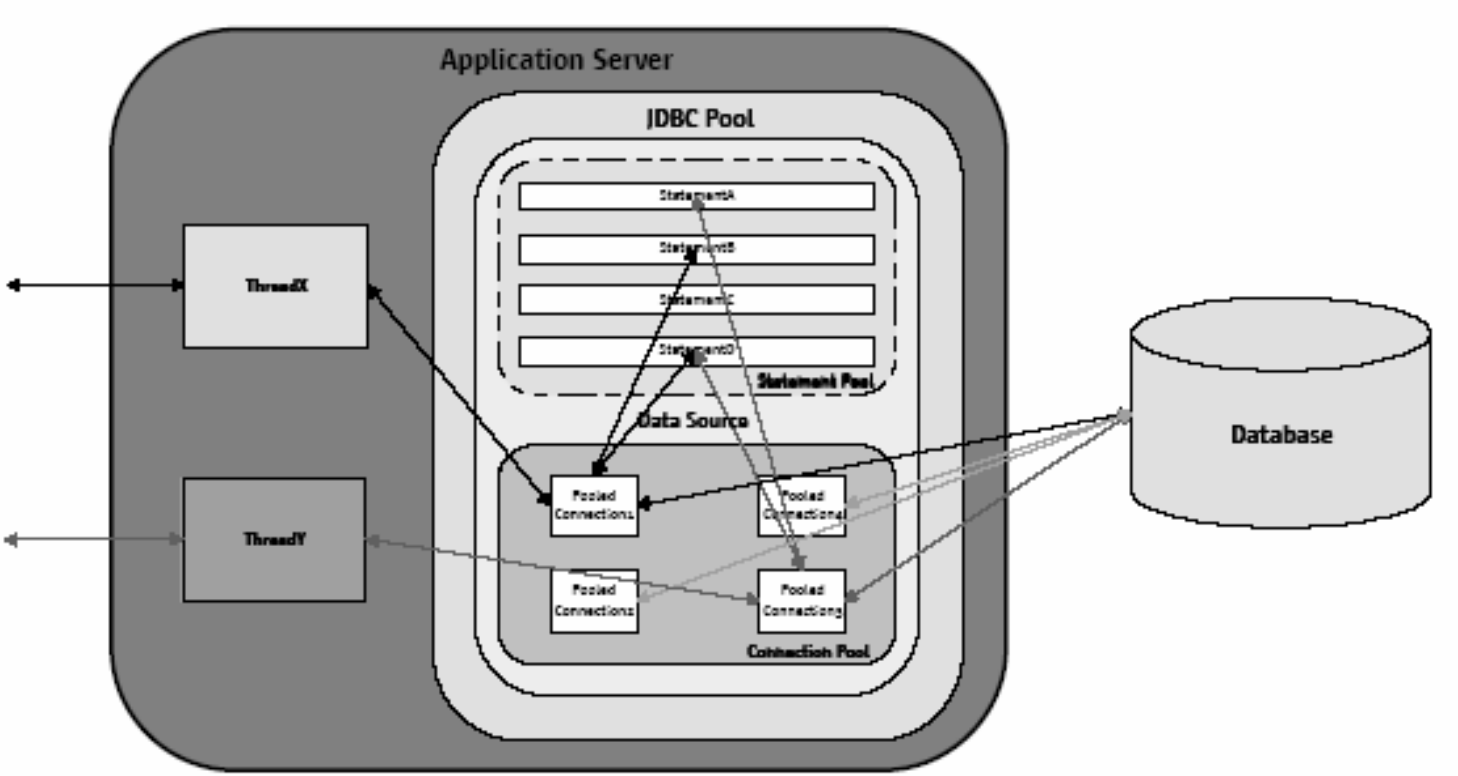
Issue	*Description*
Dirty read	TX2 reads uncommitted changes made by another TX1; if TX1 rolls back, TX2 has dirty data. E.g. TX1 attempts to book the last seat but fails on payment; TX2 didn't see the seat so thinks it's sold out.
Non-repeatable read	"Lost update" TX1 reads; TX2 reads and updates; TX1 updates and blats over TX2's update
Phantom read	TX1 gets a list of seats; TX2 adds a seat; TX1 is unaware of the update

Concurrency issues can be controlled using one of *4 transaction isolation levels* :

	Read Type		
Isolation Level	*Dirty*	*Non-Repeatable*	*Phantom*
TRANSACTION_READ_UNCOMMITTED	Y	Y	Y
TRANSACTION_READ_COMMITTED	N	Y	Y
TRANSACTION_REPEATABLE_READ	N	N	Y
TRANSACTION_SERIALIZABLE	N	N	N

JDBC Cont..

- Use Statement pooling.



JDBC Cont..

- Statements versus Prepared Statements: When to Use What?
- Tune the SQL to minimize the data returned (e.g. not 'SELECT *').
- Avoid the Usage of Metadata Methods (e.g *DatabaseMetaData.getColumns()* whenever Possible. They are expensive.
- Try to combine queries and batch updates.

JDBC Cont..

- Use stored procedures.
- Cache data to avoid repeated queries.
- Close resources (Connections, Statements, ResultSets) when finished with.
- Select the fastest JDBC driver.

http://www.oreilly.com/catalog/jorajdbc/chapter/ch19.html#74469

- Link to read:

http://www.precisejava.com/javaperf/j2ee/JDBC.htm

Why coding conventions are important?

- We all have our own coding style but our code is not a piece of art for others.
- Use most widely used conventions for method, fields naming. Refer Sun site.
- Make set of coding standards mandatory relevant to your product and project maintenance.
- Document inheritance, overload, static checked exceptions without fail.