Franklin D. Worrell
5125-001 Spring 2017
Final Project Inspection
1 May 2017

Concurrency Tests 1-4:
Initial Description of Experiments and Software

Four concurrency experiments were designed and tested. They make use of four applications: ReadReadCertificate, ReadWriteCertificate, ReadCertificate, and WriteCertificate. The first two are the processes whose transactions get interrupted and the latter two are the interrupters. Hence, as the names suggest, four scenarios are explored:

1. a read-write transaction interrupted by a read,
2. a read-write transaction interrupted by a write,
3. a read-read transaction interrupted by a read, and
4. a read-read transaction interrupted by a write.

JDBC allows for six possible transaction isolation levels, and of these, three were considered: Serializable, Read Committed, Read Uncommitted in the following experiments. The values and associated constants are given in the following table[1]:

Isolation Level Constant Name	Integer Constant
TRANSACTION_NONE	0
TRANSACTION_READ_COMMITTED	2
TRANSACTION_READ_UNCOMMITTED	1
TRANSACTION_REPEATABLE_READ	4
TRANSACTION_SERIALIZABLE	8

These integer values are important, because both ReadWriteCertificate and ReadReadCertificate take one of these integers as a command-line argument at runtime. This allows for flexibility and easy replication of the experimental procedure and outcomes.

Experiments preceded as follows, either ReadReadCertificate or ReadWriteCertificate was run in one terminal with 2, 1, or 8 provided as a command-line argument. Then, in a second terminal, either ReadCertificate or WriteCertificate was run when the program running in the first terminal prompted. The outcomes of all possible combinations are presented in what follows. To ensure a neutral starting state for each experiment, the database was cleaned up by the provided CleanUpAfterEachExperiment program. All source code is included in the src folder.

[1] Taken from https://docs.oracle.com/javase/8/docs/api/constant-values.html#java.sql. Accessed on 30 April 2017.

Concurrency Test 1:
Read-Write Transaction Interrupted with Read

Let's say that a new edition of SQL Server will be released later this year, SQL Server 2017. However, this edition of SQL Server does not make substantial changes to the language or its features—instead it contains sweeping bug fixes that neither the programmer nor the user directly see. Hence, the skills one needs to be qualified in SQL Server 2017 are the same as for SQL Server 2012 and 2014.

Microsoft offers a certification in SQL Server 2012/2014: Microsoft Certified Solutions Associate: SQL Server 2012/2014. The company decides that instead of issuing a new certificate and retiring this one, they will simply update its name (to Microsoft Certified Solutions Associate: SQL Server 2012/2014/2017). To perform the update, the person updating the database will read the value currently in certificate_name and if it has not be updated, update it. This action is analogous to querying the balance in your checking account and then withdrawing an amount based on it.

In this experiment, we interrupt the above transaction with a dirty read. Here, during the update (perhaps there is a lot of traffic on the database because it is hiring season), someone attempts to read the value in certificate_name. This experiment was performed by running ReadWriteCertificate and, when prompted, running ReadCertificate. The outcome for each of the five possible Transaction Isolation Constants that can be passed as command-line arguments to ReadWriteCertificate are given in the following table:

Isolation Level Constant Name	Integer Constant	Outcome
TRANSACTION_READ_COMMITTED	2	Success in both processes.
TRANSACTION_READ_UNCOMMITTED	1	This level not allowed.
TRANSACTION_SERIALIZABLE	8	Success in both processes.

As the table shows, some outcomes were a bit unexpected. This dirty write was permitted in both the read committed and serializable isolation levels. I found this surprising. The only explanation I can give is that no uncommitted write had yet been performed by the interrupted process; hence, the interrupting read's return value was the same as the interrupted read's. In short, since the JVM cannot see into the future, the interrupting reads were permitted because no uncommitted write had yet been performed. Additionally, I found it strange that the read uncommitted isolation level was not permitted it—it seemed as though this was a context in which it was appropriate.

Concurrency Test 2:
Read-Write Transaction Interrupted with Write

This experiment follows the exact same scheme as Test 1, with one exception: the interrupting transaction is a write instead of a read. This amounts to a process being interrupted by a blind write, which is precisely what this experiment simulates. To mirror the real-world(ish) scenario above, pretend a second person at Microsoft has decided it is time to change the name of the certificate and does so blindly, without checking it first (or notifying her/his coworkers). The procedure is the same besides the use of an interrupting writer, and the outcomes were:

Isolation Level Constant Name	Integer Constant	Outcome
`TRANSACTION_READ_COMMITTED`	2	Success in both processes.
`TRANSACTION_READ_UNCOMMITTED`	1	This level not allowed.
`TRANSACTION_SERIALIZABLE`	8	Interrupting succeeded; interrupted failed.

Again, as above, some of the outcomes were rather strange. As before, the read uncommitted level was not permitted for the application—though its utility here would be less obvious anyway. Clearly, this schedule is not serializable, so the failure there is to be expected; however, I did find it strange that is was the interrupt*ing* process that was successful in modifying the database and not the interrupted one. I would posit that this makes some sense: since the transaction in the interrupt is guaranteed to be completed, its success is less likely to result in inconsistent information than if the interrupted process was allowed to continue after the interrupting process was killed. Another strange result was that both transactions "succeeded" despite the read committed isolation level being set. This isolation level *should* not allow dirty writes. I am not sure why this occurred.

Concurrency Test 3:
Read-Read Transaction Interrupted with Read

In this third experiment, ReadReadCertificate was run and interrupted with ReadCertificate. Presumably, this scenario should not present any issues with inconsistent data. Imagine looking at the balance of both your checking and saving accounts at the ATM while your spouse looks at the balance in the savings account. Even in this real-world analogue, there are no worries about inconsistency, since no changes to the data are being made. As before, the results are summarized in the table below.

Isolation Level Constant Name	Integer Constant	Outcome
TRANSACTION_READ_COMMITTED	2	Success in both processes.
TRANSACTION_READ_UNCOMMITTED	1	This level not allowed.
TRANSACTION_SERIALIZABLE	8	Success in both processes.

Given the results of the previous two experiments, the results here are not unexpected. Since these two processes do not pose a threat to one another (in terms of invalidating each other's data), the operations are permitted on both the read committed and serializable transaction isolation level. Again, as before, the read uncommitted setting was not permitted. Still, this is strange. My guess is that Java restricts the use of this setting since it could be dangerous (if a process is interrupted several times and each interrupted is allowed to read uncommitted changes).

Concurrency Test 4:
Read-Read Transaction Interrupted with Write

In this final test, ReadReadCertificate was interrupted with WriteCertificate. In this case, there are clear worries about the consistency of the data from ReadReadCertificate's perspective. An analogous real-world situation is a person checking the balance of their household accounts while their spouse is simultaneously withdrawing from one of them. Again, the following table summarizes the results:

Isolation Level Constant Name	Integer Constant	Outcome
TRANSACTION_READ_COMMITTED	2	Both processes successfully terminate. In interrupted process, second read produces data as written by interrupting process.
TRANSACTION_READ_UNCOMMITTED	1	This level is not allowed.
TRANSACTION_SERIALIZABLE	8	Both processes successfully terminate. In the interrupted process, the values of the two reads are identical, despite change to state of database.

Once more, the inexplicable impermissibility of the read uncommitted level is a mystery. Likewise, the outcome of the processes with ReadReadCertificate running on the read committed transaction isolation level is interesting, but explicable. Since the write performed by WriteCertificate was committed while ReadReadCertificate was performing its transaction, the changed value was read during the second read operation. The interesting result was produced by ReadReadCertificate running on the serializable transaction isolation level. Here, even though the second read occurred *after* WriteCertificate had committed its change to the data, the value remained unchanged in ReadReadCertificate. I confirmed that the change by WriteCertificate had, in fact, been committed to the database. Presumably, the state surrounding a transaction occurring at the serializable isolation level is somehow held constant by the JVM so that its view into the data is not changed despite interruptions. I found it bizarre that no error was thrown by the dirty writes in these cases.