Principles of Software Construction

Serializability and Transactions

Josh Bloch  Charlie Garrod
Administrivia

- Homework 6 checkpoint due Friday 5 p.m.
- Final exam Friday, Dec 16th 5:30-8:30 p.m., GHC 4401
  - Review session Wednesday, Dec 14th 7-9:30 p.m., DH 1112
public class Max {
    public static double max(double... vals) {
        if (vals.length == 0)
            throw new IllegalArgumentException("No values!");

        double result = Double.MIN_VALUE;
        for (double val : vals)
            if (val > result)
                result = val;
        return result;
    }

    public static void main(String[] arguments) {
        System.out.println(max(-1, 0, -2.718281828));
    }
}
What does it print?

public class Max {
    public static double max(double... vals) {
        if (vals.length == 0)
            throw new IllegalArgumentException("No values!");

        double result = Double.MIN_VALUE;
        for (double val : vals)
            if (val > result)
                result = val;
        return result;
    }

    public static void main(String[] arguments) {
        System.out.println(max(-1, 0, -2.718281828));
    }
}
What does it print?

(a) 0.0
(b) 4.9E-324
(c) Throws exception
(d) None of the above

Double.MIN_VALUE is very different from Integer.MIN_VALUE
Another look

```java
public class Max {
    public static double max(double... vals) {
        if (vals.length == 0)
            throw new IllegalArgumentException("No values!");

        double result = Double.MIN_VALUE;
        for (double val : vals)
            if (val > result)
                result = val;
        return result;
    }

    public static void main(String[] arguments) {
        System.out.println(max(-1, 0, -2.718281828));
    }
}
```

**Integer.MIN_VALUE is most negative int.**

**Double.MIN_VALUE is the smallest positive double.**
You could fix it like this...

```java
public class Max {
    public static double max(double... vals) {
        if (vals.length == 0)
            throw new IllegalArgumentException("No values!");

        double result = Double.NEGATIVE_INFINITY; // Min double val
        for (double val : vals) // Min double val
            if (val > result)
                result = val;
        return result;
    }

    public static void main(String[] arguments) {
        System.out.println(max(-1, 0, -2.718281828));
    }
}
```

Prints 0.0
But this fix is much better

```java
public class Max {
    public static double max(double first, double... rest) {
        double result = first;
        for (double val : rest)
            if (val > result)
                result = val;
        return result;
    }

    public static void main(String[] arguments) {
        System.out.println(max(-1, 0, -2.718281828));
    }
}
```

Prints 0.0
The moral

- The least double val is `Double.NEGATIVE_INFINITY`, not `Double.MIN_VALUE`
  - The same is true of `Float`
- If a method requires one or more arguments, declare with `(T first, T... rest)`
  - The technique generalizes to n or more values, for any n
- For API designers
  - Don’t violate the principle of least astonishment
  - Use consistent names
Last time: MapReduce

- **Master**
  - Assign tasks to workers
  - Ping workers to test for failures

- **Map workers**
  - Map for each key/value pair
  - Emit intermediate key/value pairs

- **Reduce workers**
  - Sort data by intermediate key and aggregate by key
  - Reduce for each key

The shuffle:
MapReduce to count mutual friends and etc...

- For each pair of people in a social network, count mutual friends
  - For Map: key1 is a person, value is the list of their friends
  - For Reduce: key2 is a pair of people, values is a list of 1s, for each mutual friend that pair has

```java
f1(String key1, String value):
  for each pair of friends in value:
    EmitIntermediate(pair, 1);

f2(String key2, Iterator values):
  int result = 0;
  for each v in values:
    result += v;
  Emit(key2, result);
```

MapReduce: (person, friends)* → (pair of people, count of mutual friends)*
Today: Serializability and transactions

• A formal definition of consistency
• Introduction to transactions
• Concurrency control and serializability
• Distributed concurrency control (time permitting)
  – Two-phase commit
An aside: Double-entry bookkeeping

A style of accounting where every event consists of two separate entries: a credit and a debit

```java
void transfer(Account fromAcct, Account toAcct, int val) {
    fromAccount.debit(val);
    toAccount.credit(val);
}

static final Account BANK_LIABILITIES = ...;

void deposit(Account toAcct, int val) {
    transfer(BANK_LIABILITIES, toAcct, val);
}

boolean withdraw(Account fromAcct, int val) {
    if (fromAcct.getBalance() < val) return false;
    transfer(fromAcct, BANK_LIABILITIES, val);
    return true;
}
```
Some properties of double-entry bookkeeping

• Redundancy!
• Sum of all accounts is static
  – Can be 0
Data consistency of an application

• Suppose $\mathcal{D}$ is the database for some application and $\varphi$ is a function from database states to $\{\text{true, false}\}$
  – We call $\varphi$ an *integrity constraint* for the application if $\varphi(\mathcal{D})$ is true if the state $\mathcal{D}$ is "good"
  – We say a database state $\mathcal{D}$ is *consistent* if $\varphi(\mathcal{D})$ is true for all integrity constraints $\varphi$
  – We say $\mathcal{D}$ is inconsistent if $\varphi(\mathcal{D})$ is false for any integrity constraint $\varphi$
Data consistency of an application

- Suppose $\mathcal{D}$ is the database for some application and $\varphi$ is a function from database states to \{true, false\}
  - We call $\varphi$ an integrity constraint for the application if $\varphi(\mathcal{D})$ is true if the state $\mathcal{D}$ is "good"
  - We say a database state $\mathcal{D}$ is consistent if $\varphi(\mathcal{D})$ is true for all integrity constraints $\varphi$
  - We say $\mathcal{D}$ is inconsistent if $\varphi(\mathcal{D})$ is false for any integrity constraint $\varphi$
- E.g., for a bank using double-entry bookkeeping one possible integrity constraint is:

```python
def IsConsistent(D):
    If sum(all account balances in D) == 0:
        Return True
    Else:
        Return False
```
Database transactions

• A *transaction* is an atomic sequence of read and write operations (along with any computational steps) that takes a database from one state to another
  – "Atomic" ~ indivisible

• Transactions always terminate with either:
  – *Commit*: complete transaction's changes successfully
  – *Abort*: undo any partial work of the transaction
Database transactions

- A *transaction* is an atomic sequence of read and write operations (along with any computational steps) that takes a database from one state to another
  - "Atomic" ~ indivisible
- Transactions always terminate with either:
  - *Commit*: complete transaction's changes successfully
  - *Abort*: undo any partial work of the transaction

```java
boolean withdraw(Account fromAcct, int val) {
    begin_transaction();
    if (fromAcct.getBalance() < val) {
        abort_transaction();
        return false;
    }
    transfer(fromAcct, BANK LIABILITIES, val);
    commit_transaction();
    return true;
}
```
A functional view of transactions

- A transaction $T$ is a function that takes the database from one state $D$ to another state $T(D)$
- In a correct application, if $D$ is consistent then $T(D)$ is consistent for all transactions $T$
A functional view of transactions

- A transaction $T$ is a function that takes the database from one state $D$ to another state $T(D)$
- In a correct application, if $D$ is consistent then $T(D)$ is consistent for all transactions $T$
  - E.g., in a correct application any serial execution of multiple transactions takes the database from one consistent state to another consistent state
Database transactions in practice

• The application requests commit or abort, but the database may arbitrarily abort any transaction
  – Application can restart an aborted transaction

• Transaction ACID properties:
  – Atomicity: All or nothing
  – Consistency: Application-dependent as before
  – Isolation: Each transaction runs as if alone
  – Durability: Database will not abort or undo work of a transaction after it confirms the commit
Concurrent transactions and serializability

• For good performance, database interleaves operations of concurrent transactions
Concurrent transactions and serializability

• For good performance, database interleaves operations of concurrent transactions

• Problems to avoid:
  – Lost updates
    • Another transaction overwrites your update, based on old data
  – Inconsistent retrievals
    • Reading partial writes by another transaction
    • Reading writes by another transaction that subsequently aborts

• A schedule of transaction operations is *serializable* if it is equivalent to some serial ordering of the transactions
Concurrency control for a database

- Two-phase locking (2PL)
  - Phase 1: acquire locks
  - Phase 2: release locks
- E.g.,
  - Lock an object before reading or writing it
  - Don't release any locks until commit or abort
Summary

- Distributed systems are a great source of complexity
  - Abstractions to reduce complexity:
    - Protocols
    - RPC and computational frameworks
    - Common building blocks