Concurrency: Past and Present

Implications for Java Developers

Brian Goetz
Senior Staff Engineer, Sun Microsystems
brian.goetz@sun.com
About the speaker

• Professional software developer for 20 years
  > Sr. Staff Engineer at Sun Microsystems
• Author of *Java Concurrency in Practice*
  > Author of over 75 articles on Java development
• Member of several JCP Expert Groups
• Frequent presenter at major conferences
What I think...

Concurrency is hard.
...but don't just take my word for it

• “Unnatural, error-prone, and untestable”
• “Too hard for most programmers to use”
  > Osterhout, *Why Threads are a Bad Idea*, 1995
• “It is widely acknowledged that concurrent programming is difficult”
...but don't take their word for it

- Adding concurrency control to objects can be harder than it looks

  > Simple model of a bank account, no synchronization

  ```java
  public class Account {
      private int balance;

      public int getBalance() {
          return balance;
      }

      public void credit(int amount) {
          balance += amount;
      }

      public void debit(int amount) {
          balance -= amount;
      }
  }
  ```
Problem: Incorrect synchronization

- The Rule: if mutable data is shared between threads, *all* accesses require synchronization
  - Failing to follow The Rule is called a *data race*
  - Computations involving data races have *exceptionally subtle semantics under the Java Language Specification*
    - (that's bad)
    - Code calling `Account.credit()` could write the wrong value
  - Code calling `Account.getBalance()` could read the wrong value
Adding synchronization

• Need thread safety? Just synchronize, right?
  > It's a good start, anyway

```java
@ThreadSafe public class Account {
    @GuardedBy("this") private int balance;

    public synchronized int getBalance() {
        return balance;
    }

    public synchronized void credit(int amount) {
        balance += amount;
    }

    public synchronized void debit(int amount) {
        balance -= amount;
    }
}
```
Composing operations

• Say we want to transfer funds between accounts
  > But only if there's enough money in the account
• We can create a compound operation over multiple Accounts

```java
public class AccountManager {
    public static void transferMoney(Account from,
        Account to,
        int amount)
        throws InsufficientBalanceException {

        if (from.getBalance() < amount)
            throw new InsufficientBalanceException(...);
        from.debit(amount);
        to.credit(amount);
    }
}
```
Problem: race conditions

- A race condition is when the correctness of a computation depends on “lucky timing”
  - Often caused by atomicity failures

- Atomicity failures occur when an operation should be atomic, but is not
  - Typical pattern: Check-then-act
    ```java
    if (foo != null)       // Another thread could set
      foo.doSomething(); // foo to null
    ```
  - Also: Read-modify-write
    ```java
    ++numRequests;       // Really three separate actions
    // (even if volatile)
    ```
Race Conditions

• All data in AccountManager is accessed with synchronization
  > But still has a race condition!
    > Can end up with negative balance with some unlucky timing
      – Initial balance = 100
      – Thread A: transferMoney(me, you, 100);
      – Thread B: transferMoney(me, you, 100);

```java
public class AccountManager {
    public static void transferMoney(Account from,
                                       Account to,
                                       int amount)
        throws InsufficientBalanceException {
        // Unsafe check-then-act
        if (from.getBalance() < amount)
            throw new InsufficientBalanceException(...);
        from.debit(amount);
        to.credit(amount);
    }
}
```
Demarcating atomic operations

- Programmer must specify atomicity requirements
  > We could lock both accounts while we do the transfer
  > (Provided we know the locking strategy for Account)

```java
public class AccountManager {
    public static void transferMoney(Account from, Account to, int amount)
        throws InsufficientBalanceException {
            synchronized (from) {
                synchronized (to) {
                    if (from.getBalance() < amount)
                        throw new InsufficientBalanceException(...);
                    from.debit(amount); to.credit(amount);
                }
            }
        }
```

Problem: Deadlock

- Deadlock can occur when multiple threads each acquire multiple locks in different orders
  > Thread A: transferMoney(me, you, 100);
  > Thread B: transferMoney(you, me, 50);

```java
public class AccountManager {
    public static void transferMoney(Account from,
                                       Account to,
                                       int amount)
        throws InsufficientBalanceException {

        synchronized (from) {
            synchronized (to) {
                if (from.getBalance() < amount)
                    throw new InsufficientBalanceException(...);
                from.debit(amount);
                to.credit(amount);
            }
        }
    }
}
```
Avoiding Deadlock

• We can avoid deadlock by *inducing a lock ordering*

```java
public class AccountManager {
    public static void transferMoney(Account from,
                                           Account to,
                                           int amount)
        throws InsufficientBalanceException {

        Account first, second;
        if (from.getAccountNumber() < to.getAccountNumber()) {
            first = from; second = to;
        } else {
            first = to; second = from;
        }

        synchronized (first) {
            synchronized (second) {
                if (from.getBalance() < amount)
                    throw new InsufficientBalanceException(...);
                from.debit(amount);
                to.credit(amount);
            }
        }
    }
}
```
That was hard!

• We started with a very simple account class
  > At every step, the “obvious” attempts at making it thread-safe had some sort of problem
  > Some of these problems were subtle and nonobvious
    > And this was a trivial class!
  > Tools didn't help us
  > Runtime didn't help us
Why was that so hard?

• There is a fundamental tension between object oriented design and threads

• OO encourages you to hide implementation details
  > Good OO design encourages composition
  > But composing thread-safe objects requires knowing how they implement locking
    > So that you can participate in their locking protocols
    > So you can avoid deadlock
    > Language hides these as implementation details

• Threads graft concurrent functionality onto a fundamentally sequential execution model
  > Threads == sequential processes with shared state
Why was that so hard?

- Threads seem like a straightforward adaptation of the sequential model to concurrent systems
  - But in reality they introduce significant complexity
    - Harder to reason about program behavior
    - Loss of determinism
    - Requires greater care

- Like going from butter to lumber
Asynchrony, before threads

• Concurrency used to refer to *asynchrony*
  > Signal handlers, interrupt handlers
  > Handler interrupts program, finishes quickly, and resumes control
  > Handlers might run in a restricted execution environment
    > Might not be able to allocate memory or call some library code

• Primary motivation was to support asynchronous IO
  > Multiple IOs meant multiple interrupts – hard to write!
  > Data accessed by both interrupt handlers and foreground program required careful coordination
Asynchrony, before threads

- Consider an *asynchronous* account interface
  - Provides asynchronous get- and set-balance operations
  - (code sketch using Java syntax)

```java
public class Accounts {
    public class AccountResult {
        public Account account;
        public int balance;
    }

    public interface GetBalCallback {
        public void callback(Object context, AccountResult result);
    }

    public interface SetBalCallback {
        public void callback(Object context, AccountResult result);
    }

    public static void getBalance(Account acct, Object context, GetBalCallback callback) {
        ... }

    public static void setBalance(Account acct, int balance, Object context, SetBalCallback callback) {
        ... }
}
```
Asynchrony, before threads

• How to build a balance-transfer operation?
  > A compound operation with four steps
    > Get from-balance, get to-balance, decrease from-balance, increase to-balance
  > Each step is an asynchronous operation
    > The callback of the first step stashes the result for later use
      – And then initiates the second step
      – And so on
      – Callback of the last step triggers callback for the compound operation

```java
public class AccountTransfer {
    public interface TransferCallback {
        public void callback(Object context, TransferResult result);
    }

    public void transfer(Account from, Account to, int amount,
                           Object context, TransferCallback callback) {...}
}
```
Asynchrony, before threads

- The code for the transfer operation in C could be 200 lines of hard-to-read code!
  - 95% is “plumbing” for the async stuff
  - Error-prone coding approach
    - Coding errors show up as operations that never complete
    - Prone to memory leaks
    - Prone to cut and paste errors
  - Hard to debug
  - Error handling made things even harder
Threads to the “rescue”

- Threads promised to turn these complex asynchronous program flows into synchronous ones
  - Now the whole control flow can be in one place
  - Code got much smaller, easier to read, less error-prone
  - A huge step forward – mostly
  - Except for that pesky shared-state problem

```java
public class Accounts {
    // blue indicates blocking operations
    public static int getBalance(Account acct) { ... }
    public static void setBalance(Account acct, int balance) { ... }

    public void transfer(Account from, Account to, int amount) {
        int fromBal = getBalance(from);
        int toBal = getBalance(to);
        setBalance(from, fromBal - amount);
        setBalance(to, toBal + amount);
    }
}
```
Threads for parallelism

• Threads were originally used to simplify asynchrony
  > MP machines were rare and expensive
• But threads also offer a promising means to exploit hardware parallelism
  > Important, because parallelism is everywhere today
  > On a 100-CPU box, a sequential program sees only 1% of the CPU cycles
Hardware trends

- Clock speeds maxed out in 2003
- But Moore's Law continues
  > Giving us more cores instead of faster cores
- Result: many more programmers become concurrent programmers (maybe reluctantly)
What are the alternatives?

- Threads are just one concurrency model
  - Threads are sequential processes that share memory
  - Any program state can change at any time
  - Programmer must prevent unwanted interactions
- There are other models too (Actors, CSP, BSP, staged programming, declarative concurrency, etc)
  - May limit *what* state can change
  - May limit *when* state can change
- Limiting the timing or scope of state changes reduces unpredictable interactions
- Can improve our code by learning from other models
What are the alternatives?

• The rule in Java is
  > Hold locks when accessing shared, mutable state
  > Hold locks for duration of atomic operations

• Managing locking is difficult and error-prone

• The alternatives are
  > Don't mutate state
    > Eliminates need for coordination
  > Don't share state
    > Isolates effect of state changes
  > Share state only at well-defined points
    > Make the timing of concurrent modifications explicit
Prohibit mutation: functional languages

- Functional languages (e.g., Haskell, ML) outlaw mutable state
  - Variables are assigned values when they are declared, which never change
  - Expressions produce a value, but have no side effects
- No mutable state $\rightarrow$ no need for synchronization!
  - No races, synchronization errors, atomicity failures
- No synchronization $\rightarrow$ no deadlock!
Applying it to Java: prefer immutability

• You can write immutable objects in Java
  > And you should!
  > Functional data structures can be efficient too

• Immutable objects are automatically thread-safe
  > And easier to reason about
  > And safer
  > And scale better

• Limit mutability as much as you can get away with
  > The less mutable state, the better
  > Enforce immutability if possible
    > Final is the new private!
Explicit concurrency: message passing

- With message-passing, mutable state is private to an activity
  - Interface to that activity is via messages
  - If you want to read it, ask them for the value
  - If you want to modify it, ask them to do it for you

- This makes the concurrency explicit
  - Apart from send/receive, all code behaves sequentially
Erlang: functional + message passing

• Everything is an Actor (analogous to threads)
• Actors have an address, and can
  > Send messages to other Actors
  > Create new Actors
  > Designate behavior for when a message is received
• Concurrency is explicit – send or receive messages
  > No shared state!
• Used in telephone switches
  > 100KLoc, less than 3m/year downtime
Example: a simple counter in Erlang

- State in Erlang is local to an Actor
  > Each counter is an Actor, who owns the count
  > Clients send either “increment” or “get value” messages

```erlang
increment(Counter) ->
    Counter ! increment.  %Send “increment” to Counter actor

value(Counter) ->
    Counter ! {self(), value},  %Send (my address, “value”) tuple
    receive
        {Counter, Value} -> Value
    end.

%% The counter loop.
loop(Val) ->
    receive
        increment -> loop(Val + 1);
        {From, value} -> From ! {self(), Val}, loop(Val);
        Other -> loop(Val)  % All other messages
    end.
```

- No shared or mutable state!
Actors in Scala

• Scala is an object-functional hybrid for the JVM
  > Similar in spirit to F# for .NET
  > Scala also supports an Actor model

```scala
class OnePlaceBuffer {
  private val m = new MailBox // An internal mailbox
  private case class Empty, Full(x: Int) // Msg types
  m send Empty // Initialization
  def write(x: Int)
    { m receive { case Empty => m send Full(x) } }
  def read: Int = m receive {
    case Full(x) => m send Empty; x
  }
}

> Uses partial functions to select messages
```
Single mutation: the declarative model

- Functional languages have only bind, not assign
- The declarative concurrency model relaxes this somewhat to provide *dataflow variables*
  - Single-assignment (write-once) variables
    - Can either be unassigned or assigned
      - Only state transition is undefined → defined
    - Assigning more than once is an error
    - Reads to unassigned variables *block* until a value is assigned
- Nice: all possible executions with a given set of inputs have equivalent results
  - No races, locking, deadlocks
- Can be implemented in Java using Future classes
Responsible concurrency

- I don't expect people are going to ditch Java in favor of CSP, Erlang, or other models any time soon
- But we can try to restore predictability by limiting the nondeterminism of threads
  - Limit concurrent interactions to well-defined points
    - Encapsulate code that accesses shared state in frameworks
  - Limit shared data
    - Consider copying data instead of sharing it
  - Limit mutability
- Each of these reduces risk of unwanted interactions
  - Moves us closer to restoring determinism
Recommendations

- Concurrency is hard, so minimize the amount of code that has to deal with concurrency
  > Isolate concurrency in concurrent components such as blocking queues
  > Isolate code that accesses shared state in frameworks
- Use immutable objects wherever you can
  > Immutable objects are automatically thread safe
  > If you can't eliminate all mutable state, eliminate as much as you can
- Sometimes it's cheaper to share a non-thread-safe object by copying than to make it thread-safe
Development to watch: Software Transactional Memory (STM)

- Most promising approach for integrating with Java
  > Not here yet, waiting for research improvements
- Replace explicit locks with transaction boundaries
  ```java
  atomic {
    from.credit(amount);
    to.debit(amount);
  }
  > Explicit locking causes problems if locking granularity doesn't match data access granularity
  > Let platform figure out what state is accessed and choose the locking strategy
  > No deadlock risk
    > Conflicts can be detected and rolled back
  > Transactions compose naturally!
Concurrency: Past and Present

Implications for Java Developers

Brian Goetz
Senior Staff Engineer, Sun Microsystems
brian.goetz@sun.com