Principles of Software Construction: Objects, Design, and Concurrency

Concurrency – part 2

Synchronization, communication, and liveness

Josh Bloch    Charlie Garrod    Darya Melicher
Administrivia

- Reading due today
  - Java Concurrency in Practice, Sections 11.3 and 11.4
- Homework 5a due tomorrow
- 2nd midterm exam returned today, after class
- Oh, and one more thing...
Midterm 2 statistics

Mean raw score was 44 / 85
Problem 2 – Boolean expressions

• Many people misinterpreted this problem
• You had to write a library to **represent** Boolean expressions
  – In other words, to allow clients to create & use Boolean expression objects
• Boolean expressions were defined (recursively) as
  – The constant expressions *true* and *false*
  – The unary expression ![x], where x is a Boolean expression
  – The binary expressions x & y and x | y, where x and y are Boolean expressions
• Instances had to support two functions
  – Compute the boolean value to which the expression evaluates
    • e.g., the value of (true | false) & !(true and false) is true
  – Convert the expression into an unambiguous string representation
    • e.g., "((true | false) & !(true and false))"
• You had to support new operators without modifying library
The most common correct solution (1/3)

```java
public interface BooleanExpression {
    boolean eval();
    // You don't need toString in interface; all objects have it
}

public final class ConstantExpression implements BooleanExpression {
    private final boolean val;

    public ConstantExpression2(boolean val) {
        this.val = val;
    }

    @Override public boolean eval() {
        return val;
    }

    @Override public String toString() {
        return String.valueOf(val);
    }
}
```
The most common correct solution (2/3)

```java
public final class NotExpression implements BooleanExpression {
    private final BooleanExpression op;

    public NotExpression(BooleanExpression op) {
        this.op = Objects.requireNonNull(op);
    }

    @Override public boolean eval() {
        return !op.eval();
    }

    @Override public String toString() {
        return "!" + op.toString();
    }
}

// And a similar class for AndExpression, OrExpression
```
The most common correct solution (3/3)

```java
public final class XorExpression implements BooleanExpression {
    private final BooleanExpression op1, op2;

    public XorExpression(BooleanExpression op1, BooleanExpression op2) {
        this.op1 = Objects.requireNonNull(op1);
        this.op2 = Objects.requireNonNull(op2);
    }

    @Override public boolean eval() {
        return op1.eval() ^ op2.eval();
    }

    @Override public String toString() {
        return String.format("(%s ^ %s)", op1, op2);
    }
}
```
Critique of previous solution

• Not bad!
  – Fully solved the problem, and earned most of the points
• But could be shorter, and achieve better code reuse
• What’s the major source of code duplication?
  – Hint: would get worse if we added functionality to BooleanExpression
• How do you fix it?
• How could ConstantExpression be made (much) shorter?
Improved solution – adds binary operator interface (1/3)
Retains BooleanExpression interface and NotExpression from previous solution

```java
public enum ConstantExpression implements BooleanExpression {
    TRUE, FALSE;
    public boolean eval() { return this == TRUE; }
}

public interface BinaryOperator {
    boolean apply(boolean operand1, boolean operand2);
}

public enum BasicBinaryOperator2 implements BinaryOperator {
    AND { public boolean apply(boolean op1, boolean op2) { return op1 && op2; }},
    OR { public boolean apply(boolean op1, boolean op2) { return op1 || op2; }};
}
```
Improved solution – binary expression (2/3)

```java
public final class BinaryExpression implements BooleanExpression {
    private final BinaryOperator operator;
    private final BooleanExpression op1, op2;

    public BinaryExpression(BooleanExpression op1,
                             BinaryOperator operator, BooleanExpression op2) {
        this.operator = Objects.requireNonNull(operator);
        this.op1 = Objects.requireNonNull(op1);
        this.op2 = Objects.requireNonNull(op2);
    }

    @Override public boolean eval() {
        return operator.apply(op1.eval(), op2.eval());
    }

    @Override public String toString() {
        return String.format("(%s %s %s)", op1, operator, op2);
    }
}
```
Improved solution – Part b (3/3)

public class XorOperator implements BinaryOperator {
    private XorOperator() { }  // Singleton
    public static final XorOperator XOR = new XorOperator();

    @Override public boolean apply(boolean op1, boolean op2) {
        return op1 ^ op2;
    }

    @Override public String toString() { return "^"; }
}
Alternative BasicBinaryOperator enum

No instance-specific class bodies, nicer string form

```java
public enum BasicBinaryOperator implements BinaryOperator {
    AND((op1, op2) -> op1 & op2, "&"),
    OR((op1, op2) -> op1 | op2, "|");

    private final BinaryOperator function;
    private final String symbol;

    BasicBinaryOperator(BinaryOperator function, String symbol) {
        this.function = function;
        this.symbol = symbol;
    }

    @Override public boolean apply(boolean op1, boolean op2) {
        return function.apply(op1, op2);
    }

    @Override public String toString() { return symbol; }
}
```
Sample program and its output

```java
public class Sample {
    public static void main(String[] args) {
        BooleanExpression e1 = new BinaryExpression(TRUE, OR, FALSE);
        BooleanExpression e2 = new BinaryExpression(FALSE, OR, TRUE);
        BooleanExpression e3 = new BinaryExpression(e1, AND, e2);
        print(e3);
        BooleanExpression e4 = new NotExpression(e2);
        print(e4);
        print(new BinaryExpression(e3, XOR, e4));
    }

    private static void print(BooleanExpression e1) {
        System.out.printf("%s = %b\n", e1, e1.eval());
    }
}

((TRUE | FALSE) & (FALSE | TRUE)) = true
!(FALSE | TRUE) = false
(((TRUE | FALSE) & (FALSE | TRUE)) ^ !(FALSE | TRUE)) = true
```
The main design pattern: Composite!

- **Definition:** “Compose objects into **tree structures** to represent part-whole hierarchies. Composite **lets clients treat individual objects and compositions of objects uniformly.**”
- Boolean constants are the “individual objects”
- Operations (unary and binary) are the “Compositions”
- This is a textbook example of the pattern
- Just because it uses an interface doesn't make it strategy pattern
  - BinaryExpression / BinaryOperator solution uses strategy, too
  - But not many of you did that solution
Lessons from problem 2

• Don’t start work on a problem until you understand it
  – If you’re not sure, ask!
• If you think a solution will be long/repetitious, ask yourself:
  – What code is repeated? How could I “factor it out”?
• If a type has only a few, known, instances, consider an enum
  – Generally prefer instance-specific data to class-bodies
  – But feel free to make exceptions where it works
Problem 3 – Bus info system design

• Major design decision: should routes/stops be mutable or immutable?
• Implications of this decision permeate design.
• **All design principles suggest immutable!**
• Arguments for mutability were generally flawed
• Adding responsibility for publishing or subscription-tracking to bus route
  – **Decreases** cohesion
  – **Increases** representational gap
  – Unlikely to improve performance
  – Let a bus route be a bus route!
• Maintain a Map<BusRoute, Set<Subscriber>> (or some such)
Key concepts from last Thursday
Example: Money-grab

```java
public static void main(String[] args) throws InterruptedException {
    BankAccount bugs = new BankAccount(100);
    BankAccount daffy = new BankAccount(100);

    Thread bugsThread = new Thread(() -> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(daffy, bugs, 100);
    });

    Thread daffyThread = new Thread(() -> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(bugs, daffy, 100);
    });

    bugsThread.start(); daffyThread.start();
    bugsThread.join(); daffyThread.join();
    System.out.println(bugs.balance() + daffy.balance());
}
```
Example: Money-grab

*Broken – What’s wrong?*

```java
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static void transferFrom(BankAccount source, BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance += amount;
    }

    public long balance() {
        return balance;
    }
}
```
Outline (such as it is) for today’s lecture

I. Serial number generation example
II. Cooperative thread termination example
III. Liveness and deadlock
I. Example: serial number generation

*What would you expect this to print?*

```java
public class SerialNumber {
    private static long nextSerialNumber = 0;

    public static long generateSerialNumber() {
        return nextSerialNumber++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread threads[] = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads) thread.join();
        System.out.println(generateSerialNumber());
    }
}
```
What went wrong?

- The ++ (increment) operator is not atomic!
  - It reads a field, increments the value, and writes it back
- If multiple calls to `generateSerialNumber` see the same value, they generate duplicates
The fix is easy

```java
public class SerialNumber {
    private static int nextSerialNumber = 0;

    public static synchronized int generateSerialNumber() {
        return nextSerialNumber++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread threads[] = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads) thread.join();
        System.out.println(generateSerialNumber());
    }
}
```
But you can do better!

java.util.concurrent is your friend

public class SerialNumber {
    private static AtomicLong nextSerialNumber = new AtomicLong();

    public static long generateSerialNumber() {
        return nextSerialNumber.getAndIncrement();
    }

    public static void main(String[] args) throws InterruptedException {
        Thread threads[] = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for (Thread thread : threads) thread.join();
        System.out.println(generateSerialNumber());
    }
}
Aside: Hardware abstractions

- Supposedly:
  - Thread state shared in memory

- A (slightly) more accurate view:
  - Separate state stored in registers and caches, even if shared
Atomicity

• An action is *atomic* if it is indivisible
  – Effectively, it happens all at once
    • No effects of the action are visible until it is complete
    • No other actions have an effect during the action
• In most languages (including java) increment is *not* atomic

\[ \text{val}++; \]

is actually

1. **Load** data from field `val`
2. Increment data by 1
3. **Store** data to field `val`
Some actions are atomic

**Precondition:**

\[
\text{int } i \equiv 7;
\]

**Thread A:**

\[
i = 42;
\]

**Thread B:**

\[
\text{ans } = i;
\]

- What are the possible values for `ans`?
Some actions are atomic

Precondition:  Thread A:  Thread B:
\texttt{int }i \texttt{ = = 7;}  \texttt{i = 42;}  \texttt{ans = i;}

- What are the possible values for \texttt{ans}?  

\begin{center}
\begin{tabular}{|c|c|}
\hline
\texttt{ans: } & \texttt{00000...00000111} \\
\hline
\texttt{ans: } & \texttt{00000...00101010} \\
\hline
\end{tabular}
\end{center}
Some actions are atomic

Precondition:
\[
\text{int } i = 7;
\]

Thread A:
\[
i = 42;
\]

Thread B:
\[
\text{ans} = i;
\]

- What are the possible values for \text{ans}?

\[
\text{ans}: \ 00000\ldots00000111
\]

\[
\text{ans}: \ 00000\ldots00101010
\]

- In Java:
  - Reading an int field is atomic
  - Writing an int field is atomic

  - Thankfully, \[
  \text{ans}: \ 00000\ldots00101111
  \]
  is not possible.
Bad news: some simple actions are not atomic

- Consider a single 64-bit long value

```
<table>
<thead>
<tr>
<th>high bits</th>
<th>low bits</th>
</tr>
</thead>
</table>
```

- Concurrently:
  - Thread A writing high bits and low bits
  - Thread B reading high bits and low bits

Precondition: \( \text{long } i == 10_000_000_000; \)
Thread A: \( i = 42; \)
Thread B: \( \text{ans} = i; \)

\[
\begin{align*}
\text{ans: } & \quad 01001\ldots00000000 \quad (10,000,000,000) \\
\text{ans: } & \quad 00000\ldots00101010 \quad (42) \\
\text{ans: } & \quad 01001\ldots00101010 \quad (10,000,000,042)
\end{align*}
\]
But none of this matters if you synchronize access to any shared mutable state!
II. Another example: cooperative thread termination

public class StopThread {
    private static boolean stopRequested;

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested)
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(10);
        stopRequested = true;
    }
}
What went wrong?

- In the absence of synchronization, there is no guarantee as to when, if ever, one thread will see changes made by another!

- JVMs can and do perform this optimization:
  
  ```java
  while (!done)
      /* do something */ ;
  
  becomes:
  ```

  ```java
  if (!done)
      while (true)
          /* do something */ ;
  ```
How do you fix it?

```java
public class StopThread {
    private static boolean stopRequested;
    private static synchronized void requestStop() {
        stopRequested = true;
    }

    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested()) {
                /* Do something */
            }
            /* Do something */
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(10);
        requestStop();
    }
}
```
A better(?) solution

volatile is synchronization without mutual exclusion

```java
public class StopThread {
    private static volatile boolean stopRequested;

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested)
                /* Do something */;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(10);
        stopRequested = true;
    }
}
```
Outline for today’s lecture

I. Serial number generation example
II. Cooperative thread termination example
III. Liveness and deadlock
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }
    static synchronized void transferFrom(BankAccount source, BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance += amount;
    }
    public synchronized long balance() {
        return balance;
    }
}
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static void transferFrom(BankAccount source, BankAccount dest, long amount) {
        synchronized(BankAccount.class) {
            source.balance -= amount;
            dest.balance += amount;
        }
    }

    public synchronized long balance() {
        return balance;
    }
}
A proposed fix: *lock splitting*

```java
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static void transferFrom(BankAccount source, BankAccount dest, long amount) {
        synchronized(source) {
            synchronized(dest) {
                source.balance -= amount;
                dest.balance += amount;
            }
        }
    }

    ...
}
```
A liveness problem: deadlock

- A possible interleaving of operations:
  - bugsThread locks the daffy account
  - daffyThread locks the bugs account
  - bugsThread attempts to lock the bugs account...
  - daffyThread attempts to lock the daffy account...
A liveness problem: deadlock

```java
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static void transferFrom(BankAccount source, BankAccount dest, long amount) {
        synchronized(source) {
            synchronized(dest) {
                source.balance -= amount;
                dest.balance += amount;
            }
        }
    }
}
```
Avoiding deadlock

- The *waits-for graph* represents dependencies between threads
  - Each node in the graph represents a thread
  - An edge T1->T2 represents that thread T1 is waiting for a lock T2 owns
- Deadlock has occurred iff the waits-for graph contains a cycle
- One way to avoid deadlock: locking protocols that avoid cycles
Avoiding deadlock by ordering lock acquisition

public class BankAccount {
    private long balance;
    private final long id = SerialNumber.generateSerialNumber();

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static void transferFrom(BankAccount source, BankAccount dest, long amount) {
        BankAccount first = source.id < dest.id ? source : dest;
        BankAccount second = first == source ? dest : source;
        synchronized (first) {
            synchronized (second) {
                source.balance -= amount;
                dest.balance += amount;
            }
        }
    }
}
Another subtle problem: The lock object is exposed

```java
public class BankAccount {
    private long balance;
    private final long id = SerialNumber.generateSerialNumber();

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static void transferFrom(BankAccount source,
                              BankAccount dest, long amount) {
        BankAccount first = source.id < dest.id ? source : dest;
        BankAccount second = first == source ? dest : source;
        synchronized (first) {
            synchronized (second) {
                source.balance -= amount;
                dest.balance += amount;
            }
        }
    }
}
```
An easy fix: Use a private lock

```java
public class BankAccount {
    private long balance;
    private final long id = SerialNumber.generateSerialNumber();
    private final Object lock = new Object();

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static void transferFrom(BankAccount source, BankAccount dest, long amount) {
        BankAccount first = source.id < dest.id ? source : dest;
        BankAccount second = first == source ? dest : source;
        synchronized (first.lock) {
            synchronized (second.lock) {
                source.balance -= amount;
                dest.balance += amount;
            }
        }
    }
}
```
Concurrency and information hiding

- Encapsulate an object's state: Easier to implement invariants
  - Encapsulate synchronization: Easier to implement synchronization policy
Summary

• Like it or not, you’re a concurrent programmer
• Ideally, avoid shared mutable state
• If you can’t avoid it, synchronize properly
  – Failure to do so causes safety and liveness failures
  – If you don’t synchronize properly, your program won’t work
• Even atomic operations require synchronization
  – e.g., stopRequested = true;
  – And some things that look atomic aren’t (e.g., val++)
• If you use locks, watch out for deadlock!
  – Resource-ordering required for multiple locks