Principles of Software Construction: Objects, Design, and Concurrency

Part 42: Concurrency

Introduction to concurrency

Josh Bloch    Charlie Garrod    Darya Melicher
Administrivia

• Homework 5 team sign-up deadline tonight
  • Team sizes, presentation slots, ...
  – Midterm exam in class Thursday (November 1\textsuperscript{st})
    • Review session today 7-9 p.m. Porter Hall 100
  – Next required reading due Tuesday
    • Java Concurrency in Practice, Sections 11.3 and 11.4
  – Homework 5 frameworks discussion
Today

• Some puzzlers
• API design conclusions
• Introduction to concurrency

If you pay $2.00 for a gasket that costs $1.10, how much change do you get?

```java
public class Change {
    public static void main(String args[]) {
        System.out.println(2.00 - 1.10);
    }
}
```

From An Evening Of Puzzlers by Josh Bloch
What does it print?

(a) 0.9  
(b) 0.90  
(c) It varies  
(d) None of the above

public class Change {
    public static void main(String args[]) {
        System.out.println(2.00 - 1.10);
    }
}

What does it print?

(a) 0.9
(b) 0.90
(c) It varies
(d) None of the above: 0.8999999999999999

Decimal values can't be represented exactly by float or double
Another look

```java
public class Change {
    public static void main(String args[]) {
        System.out.println(2.00 - 1.10);
    }
}
```
How do you fix it?

// You could fix it this way...
import java.math.BigDecimal;
public class Change {
    public static void main(String args[]) {
        System.out.println(
            new BigDecimal("2.00").subtract(
                new BigDecimal("1.10")));
    }
}

// ...or you could fix it this way
public class Change {
    public static void main(String args[]) {
        System.out.println(200 - 110);
    }
}
The moral

- Avoid float and double where exact answers are required
  - For example, when dealing with money
- Use BigDecimal, int, or long instead
2. “A Change is Gonna Come”

If you pay $2.00 for a gasket that costs $1.10, how much change do you get?

```java
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}
```
What does it print?

(a) 0.9
(b) 0.90
(c) 0.8999999999999999
(d) None of the above

import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}
What does it print?

(a) 0.9
(b) 0.90
(c) 0.89999999999999999
(d) None of the above:
0.8999999999999999911182158029987476766109466552734375

We used the wrong BigDecimal constructor
Another look

The spec says:

    public BigDecimal(double val)

Translates a double into a BigDecimal which is the exact decimal representation of the double's binary floating-point value.

```java
import java.math.BigDecimal;

public class Change {
    public static void main(String args[]) {
        BigDecimal payment = new BigDecimal(2.00);
        BigDecimal cost = new BigDecimal(1.10);
        System.out.println(payment.subtract(cost));
    }
}
```
How do you fix it?

```java
import java.math.BigDecimal;

public class Change {
    public static void main(String[] args) {
        BigDecimal payment = new BigDecimal("2.00");
        BigDecimal cost = new BigDecimal("1.10");
        System.out.println(payment.subtract(cost));
    }
}
```

Prints 0.90
The moral

• Use `new BigDecimal(String)`, not `new BigDecimal(double)`
• `BigDecimal.valueOf(double)` is better, but not perfect
  – Use it for non-constant values.
• For API designers
  – Make it easy to do the commonly correct thing
  – Make it hard to misuse
  – Make it possible to do exotic things
Key concepts from last Thursday
Key design principle: Information hiding

- "When in doubt, leave it out."
Minimize mutability

• Classes should be immutable unless there's a good reason to do otherwise
  – Advantages: simple, thread-safe, reusable
    • See java.lang.String
  – Disadvantage: separate object for each value

• Mutable objects require careful management of visibility and side effects
  – e.g. Component.getSize() returns a mutable Dimension

• Document mutability
  – Carefully describe state space
Fail fast

- Report errors as soon as they are detectable
  - Check preconditions at the beginning of each method
  - Avoid dynamic type casts, run-time type-checking

```java
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
    public Object put(Object key, Object value);

    // Throws ClassCastException if this instance
    // contains any keys or values that are not Strings
    public void void save(OutputStream out, String comments);
}
```
Subtleties of information hiding

- Prevent subtle leaks of implementation details
  - Documentation
  - Lack of documentation
  - Implementation-specific return types
  - Implementation-specific exceptions
  - Output formats
  - implements Serializable
Avoid behavior that demands special processing

- Do not return null to indicate an empty value
  - e.g., Use an empty Collection or array instead
- Do not return null to indicate an error
  - Use an exception instead
Throw exceptions only for exceptional behavior

- Do not force client to use exceptions for control flow:
  ```java
  private byte[] a = new byte[CHUNK_SIZE];

  void processBuffer (ByteBuffer buffer) {
    try {
      while (true) {
        buffer.get(a);
        ...
      }
    }
    catch (BufferUnderflowException e) {
      int remaining = buffer.remaining();
      buffer.get(a, 0, remaining);
      ...
    }
  }
  ```

- Conversely, don’t fail silently:
  ```java
  ThreadGroup.enumerate(Thread[] list)
  ```
Context: The exception hierarchy in Java

- `Object`
- `Throwable`
- `Exception`
- `Exception` (unchecked)
  - `IOException`
  - `EOFException`
  - `ClassNotFoundException`
  - `…`
- `RuntimeException`
- `NullPointerException`
- `IndexOutOfBoundsException`
- `IOException`
- `EOFException`
- `ClassNotFoundException`
- `…`
Avoid checked exceptions, if possible

- Overuse of checked exceptions causes boilerplate code:

```java
try {
    Foo f = (Foo) g.clone();
} catch (CloneNotSupportedException e) {
    // This exception can't happen if Foo is Cloneable
    throw new AssertionError(e);
}
```
Don't let your output become your de facto API

- Document the fact that output formats may evolve in the future
- Provide programmatic access to all data available in string form
Don't let your output become your de facto API

- Document the fact that output formats may evolve in the future
- Provide programmatic access to all data available in string form

```java
public class Throwable {
    public void printStackTrace(PrintStream s);
    public StackTraceElement[] getStackTrace(); // since 1.4
}

public final class StackTraceElement {
    public String getFileName();
    public int getLineNumber();
    public String getClassName();
    public String getMethodName();
    public boolean isNativeMethod();
}
```
API design summary

• Accept the fact that you, and others, will make mistakes
  – Use your API as you design it
  – Get feedback from others
  – Hide information to give yourself maximum flexibility later
  – Design for inattentive, hurried users
  – Document religiously

• It takes a lot of work to make something that appears obvious
Semester overview

- Introduction to Java and O-O
- Introduction to design
  - Design goals, principles, patterns
- Designing classes
  - Design for change
  - Design for reuse
- Designing (sub)systems
  - Design for robustness
  - Design for change (cont.)
- Design case studies
- Design for large-scale reuse
- Explicit concurrency

- Crosscutting topics:
  - Modern development tools: IDEs, version control, build automation, continuous integration, static analysis
  - Modeling and specification, formal and informal
  - Functional correctness: Testing, static analysis, verification
Concurrency, motivation and primitives

• The backstory
  – Motivation, goals, problems, ...

• Concurrency primitives in Java

• Coming soon (not today):
  – Higher-level abstractions for concurrency
  – Program structure for concurrency
  – Frameworks for concurrent computation
Power requirements of a CPU

• Approx.: \textbf{Capacitance} \times \textbf{Voltage}^2 \times \textbf{Frequency}

• To increase performance:
  – More transistors, thinner wires
    • More power leakage: \textit{increase V}
  – Increase clock frequency \textit{F}
    • Change electrical state faster: \textit{increase V}

• \textit{Dennard scaling}: As transistors get smaller, power density is approximately constant...
  – ...until early 2000s

• Heat output is proportional to power input
One option: fix the symptom

- Dissipate the heat
One option: fix the symptom

- Better: Dissipate the heat with liquid nitrogen
  - Overclocking by Tom's Hardware's 5 GHz project

http://www.tomshardware.com/reviews/5-ghz-project,731-8.html
Processor characteristics over time

Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore
Concurrency then and now

• In the past, multi-threading just a convenient abstraction
  – GUI design: event dispatch thread
  – Server design: isolate each client's work
  – Workflow design: isolate producers and consumers

• Now: required for scalability and performance
We are all concurrent programmers

• Java is inherently multithreaded
• To utilize modern processors, we must write multithreaded code
• Good news: a lot of it is written for you
  – Excellent libraries exist (java.util.concurrent)
• Bad news: you still must understand fundamentals
  – ...to use libraries effectively
  – ...to debug programs that make use of them
Aside: Concurrency vs. parallelism, visualized

• Concurrency without parallelism:

• Concurrency with parallelism:
Basic concurrency in Java

• An interface representing a task
  
  ```java
  public interface Runnable {
    void run();
  }
  ```

• A class to execute a task in a thread
  
  ```java
  public class Thread {
    public Thread(Runnable task);
    public void start();
    public void join();
    ...
  }
  ```
Example: Money-grab (1)

```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;
    }
}
```
Example: Money-grab (2)

```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());
}
```
What went wrong?

• Daffy & Bugs threads had a *race condition* for shared data
  – Transfers did not happen in sequence
• Reads and writes interleaved randomly
  – Random results ensued
The challenge of concurrency control

• Not enough concurrency control: *safety failure*
  – Incorrect computation

• Too much concurrency control: *liveness failure*
  – Possibly no computation at all (*deadlock* or *livelock*)
Shared mutable state requires concurrency control

• Three basic choices:
  1. Don't mutate: share only immutable state
  2. Don't share: isolate mutable state in individual threads
  3. If you must share mutable state: \textit{limit concurrency to achieve safety}
An easy fix:

```java
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;
    }
}
```
Concurrent control with Java's *intrinsic* locks

- **synchronized (foo) { ... }**
  - Synchronizes entire block on object `foo`; cannot forget to unlock
  - Intrinsic locks are *exclusive*: One thread at a time holds the lock
  - Intrinsic locks are *reentrant*: A thread can repeatedly get same lock
Concurrent control with Java's *intrinsic* locks

- **synchronized (foo) { ... }**
  - Synchronizes entire block on object `foo`; cannot forget to unlock
  - Intrinsic locks are *exclusive*: One thread at a time holds the lock
  - Intrinsic locks are *reentrant*: A thread can repeatedly get same lock

- **synchronized on an instance method**
  - Equivalent to `synchronized (this) { ... }` for entire method

- **synchronized on a static method in class Foo**
  - Equivalent to `synchronized (Foo.class) { ... }` for entire method
Summary

- Like it or not, you’re a concurrent programmer
- Ideally, avoid shared mutable state
  - If you can’t avoid it, synchronize properly