Quiz Review on Concurrency

Spring 2014
Why use Concurrent Programming?

1. **Natural Application Structure**
   - The world is not sequential! Easier to program multiple independent and concurrent activities.

2. **Increased application responsiveness**
   - Not blocking the entire application due to blocking IO

3. **Performance from multiprocessors and multicores**
   - Parallel execution

4. **Distributed systems**
   - Single application on multiple machines
   - Client/server type or peer-to-peer systems

© Robert Miller 2009
Concurrent in Java

Java has a predefined class java.lang.Thread which provides the mechanism by which threads are created

```java
public class MyThread extends Thread {
    public void run() {
    }
}
```

However to avoid all threads having to be subtypes of Thread, Java also provides a standard interface

```java
public interface Runnable {
    public void run();
}
```

Hence, any class which wishes to express concurrent execution must implement this interface and provide the run method

Threads do not begin their execution until the start method in the Thread class is called
import java.util.*;

public class Account {
    String id;
    String password;
    int balance;

    Account(String id, String password, int balance) {
        this.id = id;
        this.password = password;
        this.balance = balance;
    }

    boolean is_password(String password) {
        return password == this.password;
    }

    int getbal() {
        return balance;
    }

    void post(int v) {
        balance = balance + v;
    }
}

import java.util.*;

public class Bank {
    HashMap<String, Account> accounts;
    static Bank theBank = null;

    private Bank() {
        accounts = new HashMap<String, Account>();
    }

    public static Bank getbank() {
        if (theBank == null)
            theBank = new Bank();
        return theBank;
    }

    public Account get(String ID) {
        return accounts.get(ID);
    }

    ...
I need to run multiple ATM machines from my program, how do I do that?
import java.util.*;
import java.io.*;

public class ATMs extends Thread {
    static final int numATMs = 4;
    static Bank bnk;
    PrintStream out;
    BufferedReader in;
    int atmnum;

    ATM(PrintStream out, BufferedReader in) {
        this.out = out;
        this.in = in;
    }

    public static void main(String[] args) {
        bnk = Bank.getbank();
        BufferedReader stdin = new BufferedReader(
            new InputStreamReader(System.in));
        ATM atm = new ATM(System.out, stdin);
        atm.run();
    }

    public void run() {
        while(true) {
            try {
                String id = in.readLine();
                String acc = bnk.get(id);
                if (acc == null) throw new Exception();
                String pass = in.readLine();
                if (acc.is_password(pass))
                    throw new Exception();
                out.print("your balance is "+ acc.getbal());
                int val = in.read();
                if (acc.getbal() + val > 0)
                    acc.post(val);
                else
                    throw new Exception();
                out.println("Invalid input, restart");
            }
            catch(InvalidInputException e) {
                out.println("Invalid input, restart");
            }
        }
    }
}

I need to run multiple ATM machines from my program, how do I do that?
import java.util.*;
import java.io.*;

public class ATMs extends Thread {
    static final int numATMs = 4;
    static Bank bnk;
    PrintStream out;
    BufferedReader in;
    int atmnum;

    ATMs(int num, PrintStream out, BufferedReader in) {
        this.out = out;
        this.in = in;
        this.atmnum = num;
    }

    public static void main(String[] args) {
        bnk = Bank.getbank();
        BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
        ATM atm = new ATM(System.out, stdin);
        atm.run();
    }
}

public void run() {
    while (true) {
        try {
            out.print("Account ID > ");
            String id = in.readLine();
            String acc = bnk.get(id);
            if (acc == null) throw new Exception();
            out.print("Password > ");
            String pass = in.readLine();
            if (!acc.is_password(pass)) throw new Exception();
            out.print("your balance is "+acc.getbal());
            out.print("Deposit or withdraw amount > ");
            int val = in.read();
            if (acc.getbal() + val > 0)
                acc.post(val);
            else
                throw new Exception();
            out.print("your balance is "+acc.getbal());
        } catch (Exception e) {
            out.println("Invalid input, restart");
        }
    }
}

I need to run multiple ATM machines from my program, how do I do that?
Multiple ATMs

```java
import java.util.*;
import java.io.*;

public class ATMs extends Thread {
    static final int numATMs = 4;
    static Bank bnk;
    PrintStream out;
    BufferedReader in;
    int atmnum;

    ATMs(int num, PrintStream out, BufferedReader in) {
        this.out = out;
        this.in = in;
        this.atmnum = num;
    }

    public static void main(String[] args) {
        bnk = Bank.getbank();
        ATMs atm[] = new ATMs[numATMs];
        for(int i=0; i<numATMs; i++){
            atm[i] = new ATMs(i, outdevice(i), indevice(i));
            atm[i].start();
        }
    }
}

public void run() {
    while(true) {
        try {
            out.print("Account ID > ");
            String id = in.readLine();
            String acc = bnk.get(id);
            if (acc == null) throw new Exception();
            out.print("Password > ");
            String pass = in.readLine();
            if (!acc.is_password(pass))
                throw new Exception();
            out.print("your balance is "+acc.getbal());
            out.print("Deposit or withdraw amount > ");
            int val = in.read();
            if (acc.getbal() + val > 0)
                acc.post(val);
            else
                throw new Exception();
            out.print("your balance is "+acc.getbal());
        } catch(Exception e) {
            out.println("Invalid input, restart");
        }
    }
}
```

I need to run multiple ATM machines from my program, how do I do that?
Activity trace

ATM 1

Account ID >

allyssa
Password >

MITROCKS
Your account balance is 1000
Deposit or Withdraw amount >

-200
Your account balance is 800

ATM 2

Account ID >

ben
Password >

6.005isthebest
Your account balance is 100
Deposit or Withdraw amount >

20
Your account balance is 120
Activity trace II

ATM 1

Account ID >

*ben*

Password >

*6170isthebest*

Your account balance is 100
Deposit or Withdraw amount >

-90

Your account balance is 10

ATM 2

Account ID >

*ben*

Password >

*6.005isthebest*

Your account balance is 100
Deposit or Withdraw amount >

-90

Your account balance is 10

100 - 90 - 90 = 10!!!
ATM 1

Your account balance is 100

out.print("your balance is "+acc.getbal());

Your account balance is 100

out.print("Deposit or withdraw amount > ");

Deposit or Withdraw amount >

-90

int val = in.read();

if (acc.getbal() + val > 0)

acc.post(val);

out.print("your balance is "+acc.getbal());

Your account balance is 10

ATM 2

Your account balance is 100

out.print("your balance is "+acc.getbal());

Your account balance is 100

out.print("Deposit or withdraw amount > ");

Deposit or Withdraw amount >

-90

int val = in.read();

if (acc.getbal() + val > 0)

acc.post(val);

out.print("your balance is "+acc.getbal());

Your account balance is 10
Activity trace II

ATM 1

void post(int v) {
    balance = balance + v;
}

ATM 2

void post(int v) {
    balance = balance + v;
}

balance

100

v

-90

balance

100

10

balance = balance + v;

10

balance = balance + v;

10

balance = balance + v;

10
Synchronization

All the interleavings of the threads are NOT acceptable correct programs.

Java provides synchronization mechanism to restrict the interleavings.

Synchronization serves two purposes:

- **Ensure safety** for shared updates
  - Avoid **race conditions**

- **Coordinate** actions of threads
  - Parallel computation
  - Event notification
Multiple threads access shared resource simultaneously

Safe only if:

- All accesses have no effect on resource,
  - e.g., reading a variable,
- or
- All accesses idempotent
  - E.g., $y = \text{sign}(a), a = a*2$;
- or
- Only one access at a time: mutual exclusion
**Safety: Example**

**“The too much milk problem”**

<table>
<thead>
<tr>
<th>Time</th>
<th>You</th>
<th>Your Roommate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Arrive home</td>
<td>Arrive home</td>
</tr>
<tr>
<td>3:05</td>
<td>Look in fridge, no milk</td>
<td>Look in fridge, no milk</td>
</tr>
<tr>
<td>3:10</td>
<td>Leave for grocery</td>
<td>Leave for grocery</td>
</tr>
<tr>
<td>3:15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive at grocery</td>
<td></td>
</tr>
<tr>
<td>3:25</td>
<td>Buy milk</td>
<td>Buy Milk</td>
</tr>
<tr>
<td>3:35</td>
<td>Arrive home, put milk in fridge</td>
<td>Arrive home, put up milk</td>
</tr>
<tr>
<td>3:45</td>
<td></td>
<td>Oh no!</td>
</tr>
<tr>
<td>3:50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Model of need to synchronize activities*

Courtesy of Emery Berger @ UMASS
Mutual Exclusion

Prevent more than one thread from accessing *critical section* at a given time

- Once a thread is in the critical section, no other thread can enter that critical section until the first thread has left the critical section.
- No interleavings of threads within the critical section
- **Serializes** access to section

```java
synchronized int getbal() {
    return balance;
}

synchronized void post(int v) {
    balance = balance + v;
}
```
int val = in.read();

if (acc.getbal() + val > 0)
    acc.post(val);

out.print("your balance is " +
        acc.getbal());
Your account balance is -80
Atomicity

Synchronized methods execute the body as an atomic unit
May need to execute a code region as the atomic unit
Block Synchronization is a mechanism where a region of code can be labeled as synchronized
The synchronized keyword takes as a parameter an object whose lock the system needs to obtain before it can continue
Example:

```java
    synchronized (acc) {
        if (acc.getbal() + val > 0)
            acc.post(val);
        else
            throw new Exception();
        out.print("your balance is " + acc.getbal());
    }
```
import java.util.*;
import java.io.*;

public class ATMs extends Thread {
    static final int numATMs = 1;
    static Bank bnk;
    PrintStream out;
    BufferedReader in;
    int atmnum;

    ATMs(int num, PrintStream out, BufferedReader in) {
        this.out = out;
        this.in = in;
        this.atmnum = num;
    }

    public static void main(String[] args) {
        bnk = Bank.getbank();
        ATMs atm[] = new ATMs[numATMs];
        for(int i=0; i<numATMs; i++){
            atm[i] = new ATMs(i, outdevice(i), indevice(i));
            atm[i].start();
        }
    }

    public void run() {
        while(true) {
            try {
                out.print("Account ID > ");
                String id = in.readLine();
                String acc = bnk.get(id);
                if (acc == null) throw new Exception();
                out.print("Password > ");
                String pass = in.readLine();
                if (!acc.is_password(pass))
                    throw new Exception();
                out.print("your balance is "+ acc.getbal());
                out.print("Deposit or withdraw amount > ");
                int val = in.read();
                synchronized (acc) {
                    if (acc.getbal() + val > 0)
                        acc.post(val);
                    else
                        throw new Exception();
                }
            }
            catch(Exception e) {
                out.println("Invalid input, restart");
            }
        }
    }
}
balance ATM 1

100
out.print("your balance is " + acc.getbal());
Your account balance is 100

100
out.print("Deposit or withdraw amount > ");
Deposit or Withdraw amount >

-90
int val = in.read();

ATM 2

out.print("your balance is " + acc.getbal());
Your account balance is 100

out.print("Deposit or withdraw amount > ");
Deposit or Withdraw amount >

-90
int val = in.read();

synchronized(acc)

if (acc.getbal() + val > 0)
    acc.post(val);
out.print("your balance is " + acc.getbal());
Your account balance is 10

synchronized(acc)

if (acc.getbal() + val > 0)
    throw new Exception()
public class ATMs extends Thread {
    static final int numATMs = 1;
    static Bank bnk;
    PrintStream out;
    BufferedReader in;
    int atmnum;

    ATMs(int num, PrintStream out, BufferedReader in) {
        this.out = out;
        this.in = in;
        this.atmnum = num;
    }

    public static void main(String[] args) {
        bnk = Bank.getBank();
        ATMs atm[] = new ATMs[numATMs];
        for(int i=0; i<numATMs; i++){
            atm[i] = new ATMs(i, outdevice(i), indevice(i));
            atm[i].start();
        }
    }

    public void run() {
        while(true) {
            try {
                out.print("Account ID > ");
                String id = in.readLine();
                String acc = bnk.get(id);
                if (acc == null) throw new Exception();
                out.print("Password > ");
                String pass = in.readLine();
                if (!acc.is_password(pass))
                    throw new Exception();
                out.print("your balance is "+acc.getbal());
                out.print("Deposit or withdraw amount > ");
                int val = in.read();
                synchronized (acc) {
                    if (acc.getbal() + val > 0)
                        acc.post(val);
                    else
                        throw new Exception();
                }
            } catch(Exception e) {
                out.println("Invalid input, restart");
            }
        }
    }
}
Synchronizing a block

```java
import java.util.*;
import java.io.*;

public class ATMs extends Thread {
    static final int numATMs = 1;
    static Bank bnk;
    PrintStream out;
    BufferedReader in;
    int atmnum;

    ATMs(int num, PrintStream out, BufferedReader in) {
        this.out = out;
        this.in = in;
        this.atmnum = num;
    }

    public static void main(String[] args) {
        bnk = Bank.getbank();
        ATMs atm[] = new ATMs[numATMs];
        for(int i=0; i<numATMs; i++){
            atm[i] = new ATMs(i, outdevice(i), indevice(i));
            atm[i].start();
        }
    }

    public void run() {
        while(true) {
            try {
                out.print("Account ID > ");
                String id = in.readLine();
                String acc = bnk.get(id);
                if (acc == null) throw new Exception();
                out.print("Password > ");
                String pass = in.readLine();
                if (!acc.is_password(pass))
                    throw new Exception();
                synchronized (acc) {
                    out.print("your balance is" + acc.getbal());
                    out.print("Deposit or withdraw amount >");
                    int val = in.read();
                    if (acc.getbal() + val > 0)
                        acc.post(val);
                    else
                        throw new Exception();
                    out.print("your balance is" + acc.getbal());
                }
            } catch(Exception e) {
                out.println("Invalid input, restart");
            }
        }
    }
}
```
Activity trace II

ATM 1

Account ID >

\textit{ben}

Password >

\textit{6.170isthebest}

\texttt{synchronized}(acc)

\texttt{out.print("your balance is "+acc.getbal());}

Your account balance is 100

\texttt{out.print("Deposit or withdraw amount > ");}

Deposit or Withdraw amount >

ATM 2

Account ID >

\textit{ben}

Password >

\textit{6.005isthebest}

\texttt{synchronized}(acc)
public boolean transfer(Account from, Account to, int val) {
    synchronized(from) {
        if (from.getbal() > val)
            from.post(-val);
        else
            throw new Exception();
    }
    synchronized(to) {
        to.post(val);
    }
}
**Account Transfers**

Allyssa wants to transfer $10 to Ben’s account
While Ben wants to also transfer $20 to Allyssa’s account

**Allyssa → Ben**

```java
synchronized(from)
if (from.getbal() > val)
from.post(-val);

synchronized(to)
Waiting for Ben’s account to be released to perform
```

**Ben → Allysa**

```java
synchronized(from)
if (from.getbal() > val)
from.post(-val);

synchronized(to)
Waiting for Allyssa’s account to be released to perform
```

**DEADLOCKED!**
Avoiding Deadlock

Cycle in locking graph = deadlock

Standard solution:
canonical order for locks

- Acquire in increasing order
- Release in decreasing order

Ensures deadlock-freedom, but not always easy to do
Dining Philosophers Problem

There are 5 philosophers sitting at a round table.

Between each adjacent pair of philosophers is a chopstick.

Each philosopher does two things: think and eat.

- The philosopher thinks for a while.
- When the philosopher becomes hungry, she stops thinking and...
  - Picks up left and right chopstick
  - He cannot eat until he has both chopsticks, has to wait until both chopsticks are available
  - When the philosopher gets the two chopsticks she eats
- When the philosopher is done eating he puts down the chopsticks and begins thinking again.
import java.io.*;
import java.util.*;

public class Philosopher extends Thread {
    static final int count = 5;
    Chopstick left;
    Chopstick right;
    int position;

    Philosopher(int position, Chopstick left, Chopstick right) {
        this.position = position;
        this.left = left;
        this.right = right;
    }
}

public static void main(String[] args) {
    Philosopher phil[] = new Philosopher[count];
    Chopstick last = new Chopstick();
    Chopstick left = last;
    for(int i=0; i<count; i++) {
        Chopstick right = (i==count-1)?last : new Chopstick();
        phil[i] = new Philosopher(i, left, right);
        left = right;
    }
    for(int i=0; i<count; i++) {
        phil[i].start();
    }
    ...
}
public void run() {
    try {

        while (true) {

            synchronized (left) {
                synchronized (right) {
                    System.out.println(times + " : Philosopher " + position + " is done eating");
                }
            }
        }
    } catch (Exception e) {
        System.out.println("Philosopher " + position + " 's meal got disturbed");
    }
}
Dining Philosophers Problem: Take II

```java
static Object table;
public void run() {
    try {
        while (true) {
            synchronized (table) {
                System.out.println(times + ":Philosopher " + position + " is done eating");
            }
        }
    } catch (Exception e) {
        System.out.println("Philosopher " + position + "'s meal got disturbed");
    }
}
```
public void run() {
    try {
        Chopstick first = (position%2 == 0)?left:right;
        Chopstick second = (position%2 == 0)?right:left;
        while (true) {
            synchronized(first) {
                synchronized(second) {
                    System.out.println(times+":Philosopher "+position+" is done eating");
                }
            }
        }
    } catch (Exception e) {
        System.out.println("Philosopher "+position+"'s meal got disturbed");
    }
}
Message Passing with Threads

Use a synchronized queue for message-passing between threads

- interface java.util.concurrent.BlockingQueue is such a queue
  - ArrayBlockingQueue is a fixed-size queue that uses an array representation
  - LinkedBlockingQueue is a growable queue (no FULL state) using a linked-list representation

There is no take transition in EMPTY state, so a thread that tries to take from an empty queue must block (wait) until it can.

- ArrayBlockingQueue is a fixed-size queue that uses an array representation
- LinkedBlockingQueue is a growable queue (no FULL state) using a linked-list representation
Making data safe

Confinement:
- don’t share the data between threads. Ensure that only one thread has access to it.

• Immutability:
- make the shared data immutable.

• Threadsafe datatype:
- use a datatype that does the coordination for you, like a BlockingQueue or a synchronized collection wrapper.

• Synchronization:
- when the data has to be shared between threads, keep two threads from accessing it at the same time
Concurrency Is Hard to Test

**Poor coverage**

- Recall our notions of coverage
  - all states, all transitions, or all paths through a state machine
- Given two concurrent state machines (with $N$ states and $M$ states), the combined system has $N \times M$ states (and many more transitions and paths)
- As concurrency increases, the state space explodes, and achieving sufficient coverage becomes infeasible

**Poor reproducibility**

- Transitions are **nondeterministic**, depending on relative timing of events that are strongly influenced by the environment
  - Delays can be caused by other running programs, other network traffic, operating system scheduling decisions, variations in processor clock speed, etc.
- Test driver can’t possibly control all these factors
- So even if state coverage were feasible, the test driver can’t reliably reproduce particular paths through the combined state machine
Lets Review Some Questions

Using the quizapp we developed in the last lecture!

- Additional feature: If the points for a question is run down to zero, the server will keep adding one point → No one will be locked out of a question

Goto: http://tinyurl.com/005jeopardy

Download the quizapp.jar file and either

- Double click on it and launch it
- At command line type java –jar quizapp.jar

Enter the following IP address in the dialogbox

- 128.30.66.84

Take the quiz!
The Java code

```java
synchronized (this) {
    this.accountBalance += depositAmount;
}
```

Does NOT demonstrate the concept of:

- A. shared memory
- B. message passing
- C. locking
- D. thread safety

Solution.

B. This uses locking (C) to ensure thread safety (D) of shared memory (A)
The Java code

class MyWindow extends JFrame {
    private final List<Integer> emptyList =
        new Collections.unmodifiableList(new ArrayList<Integer>());
}

Does NOT demonstrate the concept of:

- A. inheritance
- B. message passing
- C. immutability
- D. concurrency
- E. subclassing

Solution.

- B MyWindow inherits (A) from JFrame, i.e. a subclass (E). The list myList is immutable and has an empty immutable list (C) for concurrency (D).
The Java code

```java
assert(checkRep())
```

does NOT demonstrate the concept of:

- A) synchronization
- B) abstract data type
- C) rep invariant
- D) failing fast

**Solution**

- A, CheckRep checks the rep invariant (C) of the abstract data type (B), so a program that breaks its rep can fail fast (D)
The Java code

```java
private List<Person> committee;
```

does NOT demonstrate the concept of:

- A) representation independence
- B) information hiding
- C) static typing
- D) locking
- E) abstract data type

**Solution:**

- D. Private fields use information hiding (B) to ensure rep independence (A) of an ADT(E). Also uses information hiding to ensure rep independence from the concrete variant of the List ADT. Static typing © requires us to declare the type of committee, and allows us to restrict the type of its elements.
The Java code

```java
public class Boomerang implements Wiget<T> {

does NOT demonstrate the concept of:

- A) interfaces
- B) object oriented programming
- C) polymorphism
- D) Generics

Solution:

- Boomerang implements the interface (A) Wiget<T> which is a generic type (D). This is what object oriented programming (B) is.
Jim Bitdiddle dreams of being the next Zuck, so the ambitious high-schooler writes the following web app to manage the social currency of his peers:

Jim's friends Elisa and Tim are so impressed, they use the system to send the following payments to Jim:
- Elisa: 100, 500
- Tim: 50

If these were the only payments done and each person's payments are done from a single thread (i.e. there is one thread per person), the resulting socialKarma of Jim could NOT be:

- A) 1000
- B) 1650
- C) 1600
- D) 1550

Solution A
- With no mechanism for thread safety, payments may be lost.

```java
public class Person {
    private int socialKarma;
    private List<Person> buddies;
    public Person() {
        socialKarma = 1000;
        buddies = new ArrayList<Person>();
    }
    public void newBuddies(Person friend) {
        buddies.add(friend);
    }
    public void neverNeverEver(Person frenemy) {
        buddies.remove(frenemy);
    }
    public void pay(Person friend, int points) {
        assert (points > 0);
        if (socialKarma - points > 0) {
            socialKarma -= points;
            friend.socialKarma += points;
        }
    }
}
```
Jim Bitdiddle dreams of being the next Zuck, so the ambitious high-schooler writes the following web app to manage the social currency of his peers:

Jim's friends Elisa and Tim are so impressed, they use the system to send the following payments to Jim:
- Elisa: 100, 500
- Tim: 50

If these were the only payments done and each person's payments are done from a single thread (i.e. there is one thread per person), the resulting socialKarma of Jim could be:

- A) 1000
- B) 1650
- C) 1600
- D) 1550

**Solution B**

- This does not fix all concurrency issues, but at least all payments will be recorded

```java
public class Person {
    private int socialKarma;
    private List<Person> buddies;
    public Person() {
        socialKarma = 1000;
        buddies = new ArrayList<Person>();
    }
    public void newBuddies(Person friend) {
        buddies.add(friend);
    }
    public void neverNeverEver(Person frenemy) {
        buddies.remove(frenemy);
    }
    private synchronized int getK() {
        return socialKarma;
    }
    private synchronized void addK(int k) {
        socialKarma += k;
    }
    public void pay(Person friend, int points) {
        assert (points > 0);
        if (getK() - points > 0) {
            addK(-points);
            friend.addK(points);
        }
    }
}
```
Jim Bitdiddle dreams of being the next Zuck, so the ambitious high-schooler writes the following web app to manage the social currency of his peers:

Given an arbitrary graph of buddies and arbitrary payments between them, this program will never

- A) Deadlock
- B) Enter into an infinite recursion
- C) Have races, resulting in incorrect socialKarma
- D) Trigger the assert even when original points paid is > 0
- E) Works correctly

Solution B

- A due to Unorderd lock graph
- Never B because points will decrease to 0
- C due to No lock on this
- D due to integer division

```java
public class Person {
    private int socialKarma;
    private List<Person> buddies;
    public Person() {
        socialKarma = 1000;
        buddies = new ArrayList<Person>();
    }
    public void newBuddies(Person friend) {
        buddies.add(friend);
    }
    public void neverNeverEver(Person frenemy) {
        buddies.remove(frenemy);
    }
    public void pay(Person friend, int points) {
        assert (points > 0);
        synchronized (friend) {
            if (socialKarma - points > 0) {
                socialKarma -= points / 2;
                friend.socialKarma += points / 2;
                for (Person buddy : friend.buddies) {
                    pay(buddy, points / (2 * friend.buddies.size()));
                }
            }
        }
    }
}
```
Jim Bitdiddle dreams of being the next Zuck, so the ambitious high-schooler writes the following web app to manage the social currency of his peers:

Jim is concerned about the concurrent access to the buddies list in his implementation. Jim should fix the problem by

- A. Doing nothing
- B. Using a concurrent list, changing the allocation of buddies to buddies = Collections.synchronizedList(new ArrayList<Person>();
- C. Using synchronized on the newBuddies and neverNeverEver methods
- D. Putting all computation in the newBuddies and neverNeverEver methods inside a synchronized (buddies) { ... }

**Solution C**

- synchronizedList requires clients who iterate over the list to lock it, but pay does not. For that reason, D also does not help. But since pay obtains the lock on friend before iterating over friend.buddies, C will work

```java
class Person {
    private int socialKarma;
    private List<Person> buddies;

    public Person() {
        socialKarma = 1000;
        buddies = new ArrayList<Person>();
    }

    public void newBuddies(Person friend) {
        buddies.add(friend);
    }

    public void neverNeverEver(Person frenemy) {
        buddies.remove(frenemy);
    }

    public void pay(Person friend, int points) {
        assert (points > 0);
        synchronized (friend) {
            if (socialKarma - points > 0) {
                socialKarma -= points / 2;
                friend.socialKarma += points / 2;
                for (Person buddy : friend.buddies) {
                    pay(buddy, points / (2 * friend.buddies.size()));
                }
            }
        }
    }
}
```
Suppose you are reviewing this code for thread safety:

```java
public class C {
    public static final String[] x = new String[] { "abc" };  
    private final int y = 0;
    public synchronized double f() {
        double z = 0;
    }
}
```

Which of the following statements would be true and appropriate for an argument either in favor of or against the thread safety of this code?

- A) x is thread-confined.
- B) x is immutable.
- C) x is protected by a lock.
- D) x can be involved in a race condition.

**Solution D**

- While x is an immutable reference, and String values are immutable, the array is mutable, so we cannot argue for thread safety by immutability.
Suppose you are reviewing this code for thread safety:

```java
public class C {
    public static final String[] x = new String[] { "abc" }; // A) y is thread-confined.
    private final int y = 0; // B) y is immutable.
    public synchronized double f() {
        double z = 0; // C) y is protected by a lock.
    }
    private final int y = 0; // D) y can be involved in a race condition.
}
```

Which of the following statements would be true and appropriate for an argument either in favor of or against the thread safety of this code?

- A) y is thread-confined.
- B) y is immutable.
- C) y is protected by a lock.
- D) y can be involved in a race condition.

**Solution B**

- y is immutable
Suppose you are reviewing this code for thread safety:

```java
public class C {
    public static final String[] x = new String[] { "abc" };  
    private final int y = 0;
    public synchronized double f() {
        double z = 0;
    }
}
```

Which of the following statements would be true and appropriate for an argument either in favor of or against the thread safety of this code?

- A) z is thread-confined.
- B) z is immutable.
- C) z is protected by a lock.
- D) z can be involved in a race condition.

**Solution A**

- Since z is a non-final primitive, even threads spun in f cannot reference the value of z. Since f is synchronized, it is true that all access to z occurs under a lock on this, but that is not an appropriate argument for its thread safety.