```java
public class ExperimentFix1 {
    private static volatile int MY_INT = 0;

    public static void main(String[] args) throws InterruptedException
    {
        new ChangeListener().start();
        System.out.println(“wait for 2 sec");
        Thread.sleep(2000);
        new ChangeMaker().start();
    }

    static class ChangeListener extends Thread {
        public void run() {
            int local_value = MY_INT;
            while ( local_value < 5){
                if( local_value!= MY_INT){
                    System.out.println("Got Change for MY_INT : "+ MY_INT);
                     local_value= MY_INT;
                }
            }
        }
    }

    static class ChangeMaker extends Thread{
        public void run() {

            int local_value = MY_INT;
            while (MY_INT <5){
                System.out.println("Incrementing MY_INT to " + (local_value + 1));
                MY_INT = ++local_value;
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) { e.printStackTrace(); }
            }
        }
    }
}
```

```java
public class ExperimentFix2 {
    private static int MY_INT = 0;
    private static final Object obj = new Object();

static class ChangeListener extends Thread {
        public void run() {
              int local_value;
            synchronized (obj) {
              local_value = MY_INT;
            }
            while ( local_value < 5){
                synchronized (obj) {
                if( local_value!= MY_INT){
                    System.out.println("Got Change for MY_INT : "+ MY_INT);
                     local_value= MY_INT;
                }
                }
            }
        }
    }

    static class ChangeMaker extends Thread{
        public void run() {

            int local_value = MY_INT;
            while (MY_INT <5){
                System.out.println("Incrementing MY_INT to " + (local_value + 1));
                synchronized (obj) {
                            MY_INT = ++local_value;
                }
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) { e.printStackTrace(); }
            }
        }
    }
}
```

```java
import java.util.concurrent.*;

public class TimedTestConcMap extends AddRemoveTest {
    private BarrierTimer timer = new BarrierTimer();

    public TimedTestConcMap(int cap, int pairs, int trials) {
        super(cap, pairs, trials);
        barrier = new CyclicBarrier(nPairs * 2 + 1, timer);
    }

    public void test() {
        try {
            timer.clear();
            for (int i = 0; i < nPairs; i++) {
                pool.execute(new AddRemoveTest.AddWorker());
                pool.execute(new AddRemoveTest.RemoveWorker());
            }
            barrier.await();
            barrier.await();
            long nsPerItem = timer.getTime() / (nPairs * (long)
nTrials);
            System.out.print("Throughput: " + nsPerItem + "
ns/item");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) throws Exception {
        int tpt = 100000; // trials per thread
        for (int cap = 1; cap <= 1000; cap *= 10) {
            System.out.println("Capacity: " + cap);
            for (int pairs = 1; pairs <= 128; pairs *= 2) {
                  TimedTestConcMap t = new TimedTestConcMap(cap,
pairs, tpt);
                System.out.print("Pairs: " + pairs + "\t");
                t.test();
                System.out.print("\t");
                Thread.sleep(1000);
                t.test();
                System.out.println();
                Thread.sleep(1000);
            }
        }
        PutTakeTest.pool.shutdown();
    }
}
```

```java
import java.util.concurrent.*;

public class TimedTestSyncMap extends AddRemoveTest {
    private BarrierTimer timer = new BarrierTimer();

    public TimedTestSyncMap(int cap, int pairs, int trials) {
        super(cap, pairs, trials);
        barrier = new CyclicBarrier(nPairs * 2 + 1, timer);
    }

    public void test() {
        try {
            timer.clear();
            for (int i = 0; i < nPairs; i++) {
                pool.execute(new AddRemoveTest.AddWorker());
                pool.execute(new AddRemoveTest.RemoveWorker());
            }
            barrier.await();
            barrier.await();
            long nsPerItem = timer.getTime() / (nPairs * (long)
nTrials);
            System.out.print("Throughput: " + nsPerItem + "
ns/item");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) throws Exception {
        int tpt = 100000; // trials per thread
        for (int cap = 1; cap <= 1000; cap *= 10) {
            System.out.println("Capacity: " + cap);
            for (int pairs = 1; pairs <= 128; pairs *= 2) {
                  TimedTestSyncMap t = new TimedTestSyncMap(cap,
pairs, tpt);
                System.out.print("Pairs: " + pairs + "\t");
                t.test();
                System.out.print("\t");
                Thread.sleep(1000);
                t.test();
                System.out.println();
                Thread.sleep(1000);
            }
        }
        PutTakeTest.pool.shutdown();
    }
}
```

```
public class CacheV3 {
   private final ConcurrentHashMap<Integer, Future<List<Integer>>>
results = new ConcurrentHashMap<Integer, Future<List<Integer>>>();
   //the last factors must be the factors of the last number
   public List<Integer> service (final int input) throws Exception {
           Future<List<Integer>> f = results.get(input);

           if (f == null) {// if not in results table, set up Callable
to factor the number.
                   Callable<List<Integer>> eval = new
Callable<List<Integer>>() {
                            public List<Integer> call () throws
InterruptedException {
                                     return factor(input);
                            }
                  };
           FutureTask<List<Integer>> ft = new
FutureTask<List<Integer>>(eval);
           f = results.putIfAbsent(input, ft);
      // so that you won't put same key to map twice
           if (f == null) {
                   f = ft;
                   ft.run(); }
           }
         return f.get();    // waits until result is in
}
   public List<Integer> factor(int n) {
          List<Integer> factors = new ArrayList<Integer>();
          for (int i = 2; i <= n; i++) {
                  while (n % i == 0) {
                 factors.add(i);
                 n /= i;
             }
          }
         return factors;
   }
}
```

```
public class SPMDIntegration {
   public static void main (String[] args) throws Exception {
           int NTHREADS = 5;
           ExecutorService exec =
Executors.newFixedThreadPool(NTHREADS-1);
           double pi = 0;
           final double stepSize = 1.0/NTHREADS;
           for (int i = 0; i < NTHREADS; i++) {
                  final double a = i*stepSize;
                  final double b = (i+1)*stepSize;
                  Future<Double> result = exec.submit(new
Callable<Double> () {
                          public Double call() throws Exception
{
                                  return integrate(a, b);
                          }
                  });
                  pi += result.get();
         }
                  System.out.println(pi);
                  exec.shutdown();
                  exec.awaitTermination(10000,
TimeUnit.MILLISECONDS);
         }

         public static double f(double x) {
                  return 4.0/(1+x*x);
         }

         //this is the sequential program which does numerical
integration using Trapezoidal rule.
         public static double integrate(double a, double b) {
                  int N = 10000; //preciseness parameter
                  double h = (b - a) / (N - 1);     // step size
                  double sum = 1.0 / 2.0 * (f(a) + f(b));
                  for (int i = 1; i < N - 1; i++) {
                          double x = a + h * i;
                      sum += f(x);
                  }
                  return sum * h;
         }
}
```

Observed problem:
`System.out.println(cal2.after(cal1));` // should return true but it returns false

Documentation of Calendar class's after() method:

```
public boolean after(Object when) {
  return when instanceof Calendar && compareTo((Calendar) when) > 0;
}
```

In the original code, the two calendars are compared using the subclass's `after()` method which invokes the superclass's `after()` method after the `if` condition (at the else statement). However, the superclass's `after()` method would call the `compareTo()` method which is delegated to the subclass's `compareTo()` method. As a result, the subclass's `after()` method would call it's overridden `compareTo()` method. The overridden `compareTo()` method would return 0 and thus the superclass's `after()` method would return false when we expect it to return true.

In the Exercise4Fixed.java, we have a forwarder class (`ForwardingCalendar`) and its methods redirects to methods of `CalendarImplementation` class, which is a class that extends Calendar. The `CompositeCalendar` is a wrapper class that provides the same overridden methods found in the `CalendarSubclass`.

When we call the overriden `after()` method in `CompositeCalendar`, we use the `CalendarImplementation` class's `compareTo()` method. Using `super.after(when)` forwards to `ForwardingCalendar`, which invokes `CalendarImplementation's after()`. Hence, `java.util.Calendar.after()` invokes `CalendarImplementation.compareTo()` method. As a result, we won't get the same problem encountered in Exercise4.java.

```
BlockingQueue<File> queue = new LinkedBlockingQueue<File>(BOUND);
```

```java
public class StripedMapWithSize {
         // Synchronization policy: buckets[n] guarded by
locks[n%N_LOCKS]
         private static final int N_LOCKS = 16;
    private final Node[] buckets;
    private final Object[] locks;

    class Node {
        Node next;
        Object key;
        Object value;
        Node(Object key, Object value, Node next) {
            this.next = next;
            this.key = key;
            this.value = value;
        }
    }

    public StripedMapWithSize (int numBuckets) {
        buckets = new Node[numBuckets];
        locks = new Object[N_LOCKS];
        for (int i = 0; i < N_LOCKS; i++) {
            locks[i] = new Object();
        }
    }

    private final int hash(Object key) {
        return Math.abs(key.hashCode() % buckets.length);
    }

    public Object get(Object key) {
        int hash = hash(key);
        synchronized (locks[hash % N_LOCKS]) {
            for (Node m = buckets[hash]; m != null; m = m.next)
                if (m.key.equals(key))
                    return m.value;
        }
        return null;
    }

    public Object put(Object key, Object value) {
        int hash = hash(key);
        synchronized (locks[hash % N_LOCKS]) {
            for (Node m = buckets[hash]; m != null; m = m.next)
                if (m.key.equals(key)) {
                    m.value = value;
                    return m.value;
                }
            buckets[hash] = new Node(key,value,buckets[hash]);
        }
        return null;
    }

    public void clear() {
        for (int i = 0; i < buckets.length; i++) {
            synchronized (locks[i % N_LOCKS]) {
                buckets[i] = null;
            }
        }
    }

    // will not be completely accurate: while counting up the
//total, other threads might be changing the parts of the table
already viewed
    public int size() {
        int num = 0;
        for (int i = 0; i < buckets.length; i++) {
            synchronized (locks[i % N_LOCKS]) {
                for (Node m = buckets[i]; m != null; m = m.next)
                    num++;
            }
        }
        return num;
    }

    public int size2() {
        int num = 0;

        for (int i = 0; i < N_LOCKS; i++) {
         synchronized (locks[i]) {
                   for (int j = 0; j < buckets.length; j++) {
                            if (j%N_LOCKS == i) {
                                     for (Node m = buckets[i]; m
!= null; m = m.next)
                            num++;
                            }
                   }
         }
        }

        return num; }   }
```

```python
import random
DEPTH = 3
def grammar_fuzzing():
    return grammar_expr(DEPTH)     # calling for first time

def grammar_expr(depth):
    rules_expr = ["+","-", ""]
    string = ""
    randomi = random.randint(0,2)
    if randomi != 2 and depth > 1:
        string += grammar_expr(depth-1)
        string += rules_expr[randomi]
        string += grammar_term(DEPTH) #calling for first time
    else:
        string += grammar_term(DEPTH)
    return string

def grammar_term(depth):
    rules_term = ["*", "/", ""]
    string = ""
    randomi = random.randint(0,2)
    if randomi != 2 and depth > 1:
        string += grammar_term(depth-1)
        string += rules_term[randomi]
        string += grammar_factor(depth-1)
    else:   # only factor
        string += grammar_factor(depth-1)
    return string

def grammar_factor(depth):
#     EG. (-4) although not stated in the grammar rules,
#     If a factor is -Int, it should be wrapped in brackets
    rules_factor = ["-", "()", "", "."]
    string = ""
    randomi = random.randint(0,3)
    if randomi == 0:
        string += "(-"
        string += grammar_integer(DEPTH)
        string += ")"
    elif randomi == 2:   # -Int or Int
        string += rules_factor[randomi]
        string += grammar_integer(DEPTH)
    elif randomi == 1: #()
        string += "("
        string += grammar_expr(depth)
        string += ")"
    else:   # Int.Int
        string += grammar_integer(DEPTH)
        string += "."
        string += grammar_integer(DEPTH)
    return string

def grammar_integer(depth):
    string = ""
    randomi = random.randint(0,1)
    if randomi == 0:   #just digit
        return grammar_digit(False)
    else:
        string += grammar_digit(True)
        string += grammar_integer(depth-1)
        return string

def grammar_digit(boolean):
#     If grammar_digit is used for grammar_integer, the first
digit should not be zero
#     When boolean=True, grammar_digit will not return zero
    if boolean:
        return str(random.randint(1,9))
    randomi = random.randint(0,9)
    return str(randomi)

print(grammar_fuzzing())
```

```
import random

INPUT_FILE_PATH = "Testing.txt"
OUTPUT_FILE_PATH = "outputfromfuzzer.txt"

def swap(line):
#     This function takes in a line and choose a random index
#     Then, swaps the letter at this index with the letter right after
    string = ""
    if (len(line) > 1):
        index = random.randint(0, len(line)-2) # ignore the last letter
        string = line[:index] + line[index+1] + line[index] + 
line[index+2:]
         return string
    else:
        # if only one letter, return without swapping
         return line


def bitflip(line):
#     This function takes in a line
#     Then, flips a random bit of a random character in the input string
    string = ""
    letterindex = random.randint(0, len(line)-1)
    bitindex = random.randint(0, 6)
    random_letter = line[letterindex]
    # convert to binary
    binary = bin(ord(random_letter))[2:]    # get rid of '0b' in front
    if (len(binary) == 6):
        binary = "0" + binary
    if (binary[bitindex] == "1"):
        # flip 1 to 0
        binary = binary[:bitindex] + '0' + binary[bitindex+1:]
    elif (binary[bitindex] == "0"):
        # flip 0 to 1
        binary = binary[:bitindex] + '1' + binary[bitindex+1:]
    # convert binary back to character and insert to position
    string = line[:letterindex] + chr(int(binary,2)) + 
line[letterindex+1:]
         return string


def trim(line):
    string = ""
    if (len(line) > 1):
        index = random.randint(0, len(line)-2)
        string = line[:index]
         return string
    else:
        # if only one letter, return without trimming
         return line

MUTATIONS = [swap, bitflip, trim]

# main function
# input: String, file path
def generalised_fuzzer():
    file = open(INPUT_FILE_PATH)
    lines = file.read().splitlines()        # split by "\n"
    file.close()
    f = open(OUTPUT_FILE_PATH,"w+")
    for line in lines:
        output_line = random.choice(MUTATIONS)(line) # runs the random 
mutation functions
        f.write(output_line + "\n")
    f.close()


generalised_fuzzer()

# open the output file and print results
print("\n\nReading the output file")
file = open(OUTPUT_FILE_PATH)
lines = file.read().splitlines()
print(lines)
```

```
public class AddRemoveTest {
    protected static final ExecutorService pool = 
Executors.newCachedThreadPool();
    protected CyclicBarrier barrier;
    protected final Map<Integer, Integer> map;
    protected final int nTrials, nPairs;

    public static void main(String[] args) throws Exception {
        new AddRemoveTest(10, 10, 100000).test(); // sample 
parameters
        pool.shutdown();
    }

    public AddRemoveTest(int capacity, int npairs, int ntrials) {
        this.map = Collections.synchronizedMap(new 
HashMap<Integer, Integer>());
        this.nTrials = ntrials;
        this.nPairs = npairs;
        this.barrier = new CyclicBarrier(npairs * 2 + 1);
    }

    void test() {
        try {
            for (int i = 0; i < nPairs; i++) {
                pool.execute(new AddWorker());
                pool.execute(new RemoveWorker());
            }
            barrier.await(); // wait for all threads to be ready
            barrier.await(); // wait for all threads to finish
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    static int xorShift(int y) {
        y ^= (y << 6);
        y ^= (y >>> 21);
        y ^= (y << 7);
        return y;
    }

    class AddWorker implements Runnable {
        public void run() {
            try {
                int seed = (this.hashCode() ^ (int) 
System.nanoTime());
                barrier.await();
                for (int i = nTrials; i > 0; --i) {

                    map.put(seed, seed);
                    seed = xorShift(seed);
                }
                barrier.await();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }

    class RemoveWorker implements Runnable {
        public void run() {
            try {
                int seed = (this.hashCode() ^ (int) 
System.nanoTime());
                barrier.await();
                for (int i = nTrials; i > 0; --i) {
                   Object[] keys = map.keySet().toArray();
                   if (keys.length > 0) {

         map.remove(keys[Math.abs(seed)%keys.length]);

                   }
                    seed = xorShift(seed);
                }
                barrier.await();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

```
import java.util.concurrent.Phaser;
public class ExamExample
{
         private final static int numberofstudents = 2;
    public static void main(String[] args) throws
InterruptedException
    {
                     Phaser phaser = new Phaser();
                     Examiner examiner = new Examiner(phaser);
                     for (int i = 0; i < numberofstudents; i++) {
                              new Student(phaser).start();

                     }

                     examiner.start();
    }
}

class Examiner extends Thread {
         private Phaser phaser;
    public Examiner (final Phaser phaser) {
         this.phaser = phaser;
         this.phaser.register();
    }

         public void run()
         {
                   System.out.println("examiner waiting for
students to get ready;");
                   phaser.arriveAndAwaitAdvance();
                   phaser.arriveAndAwaitAdvance();
                   System.out.println("exam has ended");
         }
}

class Student extends Thread {
         private Phaser phaser;
    public Student (final Phaser phaser) {
         this.phaser = phaser;
         this.phaser.register();
    }

    public void run()
         {
                   System.out.println("student " +
Thread.currentThread().getId() + " ready;");
                   phaser.arriveAndAwaitAdvance();
                   System.out.println("exam has started" +
phaser.getPhase() + " " + phaser.getRegisteredParties());
                   System.out.println("student " +
Thread.currentThread().getId() + " taking exam;");
                   System.out.println("student " +
Thread.currentThread().getId() + " handing in exam;");
                   phaser.arrive();
                   System.out.println("student " +
Thread.currentThread().getId() + " leaves");
         }
}
```

```
public class CountDownLatchExercise {

         public static void main(String args[]) throws
InterruptedException {
                   int limit = 7;
                   final int noOfSearcher = 4;
             final CountDownLatch latch = new
CountDownLatch(limit);
             final CountDownLatch finish = new
CountDownLatch(limit);
             String[] array = new String[]{"A","B","F”,"F"};
             final Searcher[] searchers = new
Searcher[noOfSearcher];

             for (int i = 0; i < noOfSearcher; i++) {
                   searchers[i] = new Searcher(array,
i*array.length/noOfSearcher, (i+1)*array.length/noOfSearcher,
latch, finish);
                   searchers[i].start();
             }

             Thread awaitThread = new Thread( new Runnable () {
                            public void run() {
                           try {
                            System.out.println ("awaitThread
awaiting");
                                              latch.await();
                            System.out.println ("awaitThread
finishing");
                                     } catch
(InterruptedException e) {

         e.printStackTrace();
                                     }  //main thread is waiting
on CountDownLatch to finish

                                for (int i = 0; i < noOfSearcher;
i++) {
                                     searchers[i].interrupt();
                                }
                                     while (finish.getCount() >
0) {
                                              finish.countDown();
                                     }
                            }
             }
             );
             awaitThread.start();

         System.out.println ("main Thread awaiting");
                   finish.await();
         System.out.println ("main Thread finishing");
                   while (latch.getCount() > 0) {
                            latch.countDown();
                   }
         }
}

class Searcher extends Thread {
         private final String[] data;
         private final int start;
         private final int end;
         private final CountDownLatch latch;
         private final CountDownLatch finish;

         public Searcher (String[] data, int start, int end,
CountDownLatch latch, CountDownLatch finish) {
                   this.data = data;
                   this.start = start;
                   this.end = end;
                   this.latch = latch;
                   this.finish = finish;
         }

         public void run() {
             for (int i = start; i < end; i++) {
                   if (data[i].equals("F")) {
                            latch.countDown();
                   }

                   if (this.isInterrupted()) {
                   System.out.println ("interrupted");

                            break;
                   }
             }

             finish.countDown();
         }
}
```

```java
package Week10;

import java.util.Random;

public class DiningPhilFixed {
        private static int N = 5;

        public static void main (String[] args) throws Exception
{
                PhilosopherFixed[] phils = new
PhilosopherFixed[N];
                Fork[] forks = new Fork[N];

                for (int i = 0; i < N; i++) {
                        forks[i] = new Fork(i);
                }

                for (int i = 0; i < N; i++) {
                        phils[i] = new PhilosopherFixed (i,
forks[i], forks[(i+N-1)%N]);
                        phils[i].start();
                }
        }
}

class PhilosopherFixed extends Thread {
        private final int index;
        private final Fork left;
        private final Fork right;

        public PhilosopherFixed (int index, Fork left, Fork
right) {
                this.index = index;
                this.left = left;
                this.right = right;
        }

        public void run() {
                Random randomGenerator = new Random();
                try {
                        while (true) {


        Thread.sleep(randomGenerator.nextInt(100)); //not
sleeping but thinking
                                System.out.println("Phil " +
index + " finishes thinking.");
                                //with the following, there
is a global ordering on the locking
                                if (index == 0) {
                                        right.pickup();

        System.out.println("Phil " + index + " picks up right
fork.");
                                        left.pickup();

        System.out.println("Phil " + index + " picks up left
fork.");
                                }
                                else {
                                        left.pickup();

        System.out.println("Phil " + index + " picks up left
fork.");
                                        right.pickup();

        System.out.println("Phil " + index + " picks up right
fork.");
                                }

        Thread.sleep(randomGenerator.nextInt(100)); //eating
                                System.out.println("Phil " +
index + " finishes eating.");
                                left.putdown();
                                System.out.println("Phil " +
index + " puts down left fork.");
                                right.putdown();

                                System.out.println("Phil " +
index + " puts down right fork.");
                        }
                } catch (InterruptedException e) {
                        System.out.println("Don't disturb me
while I am sleeping, well, thinking.");
                }
        }
}
```

```java
public class DiningPhilFixed2 {
        private static int N = 5;
        public static void main (String[] args) throws Exception
{
                PhilosopherFixed2[] phils = new
PhilosopherFixed2[N];
                MyFork[] forks = new MyFork[N];

                for (int i = 0; i < N; i++) {
                        forks[i] = new MyFork(i);
                }

                for (int i = 0; i < N; i++) {
                        phils[i] = new PhilosopherFixed2 (i,
forks[i], forks[(i+N-1)%N]);
                        phils[i].start();
                }
        }
}
class PhilosopherFixed2 extends Thread {
        private final int index;
        private final MyFork left;
        private final MyFork right;

        public PhilosopherFixed2 (int index, MyFork left, MyFork
right) {
                this.index = index;
                this.left = left;
                this.right = right;
        }

        public void run() {
                Random randomGenerator = new Random();
                try {
                        while (true) {


        Thread.sleep(randomGenerator.nextInt(100)); //not
sleeping but thinking
                                System.out.println("Phil " +
index + " finishes thinking.");
                                if(!left.pickup()) {
                                        break;
                                }

                                System.out.println("Phil " +
index + " picks up left fork.");

                                if (!right.pickup()) {
                                        left.putdown();
                                        break;
                                }

                                System.out.println("Phil " +
index + " picks up right fork.");

        Thread.sleep(randomGenerator.nextInt(100)); //eating
                                System.out.println("Phil " +
index + " finishes eating.");
                                left.putdown();
                                System.out.println("Phil " +
index + " puts down left fork.");
                                right.putdown();

                                System.out.println("Phil " +
index + " puts down right fork.");
                        }
                } catch (InterruptedException e) {
                        System.out.println("Don't disturb me
while I am sleeping, well, thinking.");
                }
        }
}

class MyFork {
        private final int index;
        private boolean isAvailable = true;
        private final ReentrantLock reentrantLock = new
ReentrantLock();

        public MyFork (int index) {
                this.index = index;
        }

        public boolean pickup () throws InterruptedException {
                return reentrantLock.tryLock(1000,
TimeUnit.MILLISECONDS);
        }

        public void putdown() throws InterruptedException {
                reentrantLock.unlock();
        } }
```