Directions: Exam is closed book, closed notes. Answer every question; write solutions in spaces provided. Use backs of pages if necessary, but clearly indicate when this is the case. By writing your name above, you pledge to abide by the University’s Honor Code:

“I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination.”

Good luck!

Please do not write below this line.

<table>
<thead>
<tr>
<th>1.</th>
<th>2.</th>
<th>3.</th>
<th>4.</th>
<th>5.</th>
<th>6.</th>
<th>7.</th>
<th>8.</th>
<th>9.</th>
<th>10.</th>
</tr>
</thead>
</table>

SCORE ________
1. (10 points) Multiple choice. Circle your answers. Each question is worth two points.

**A. Nondeterminism arises in multi-threaded applications because:**
(a) Programmers are often unsure of how to debug such programs.
(b) Multi-threaded programs often use locks, which increase the number of possible executions a program can have.
(c) The precise order in which the different threads’ operations are performed can differ from one system run to the next.
(d) Threads are objects, and objects are by nature nondeterministic.

**B. Effectively immutable objects differ from immutable objects because:**
(a) Effectively immutable objects have all their fields declared as private and final, while immutable objects do not.
(b) Immutable objects have all their fields declared as private and final, while effectively immutable objects do not.
(c) Effectively immutable objects have only private fields, and hence are more effective than immutable objects, which may have public fields.
(d) There is no difference between effectively immutable and immutable objects.

**C. Lock-free data structures are often preferred over data structures that use locking because:**
(a) Locking can introduce significant performance penalties and unnecessary limitations on concurrent access.
(b) Correct, lock-free implementations are easier to develop and understand.
(c) Intrinsic locks in Java are not re-entrant, meaning that it is very easy to have deadlocks in locking-based implementations.
(d) None of the above.

**D. It is good practice to avoid starting a thread inside the constructor of an object because:**
(a) Starting threads inside a constructor can cause the Java Virtual Machine to crash.
(b) Starting a thread inside a constructor potentially exposes the object to another thread before the object’s construction is complete.
(c) Starting a thread inside a constructor will result in only that thread having access to the object being constructed.
(d) Constructors do not really engage in computation; they only create objects. Any thread started in a constructor will thus never begin executing.
E. The Producer / Consumer pattern is characterized by:

(a) The use of a shared stack by producers and consumers.
(b) The use of a network by producers and consumers.
(c) The use of a separate JVM by producers and consumers.
(d) The use of a shared blocking queue by producers and consumers.
2. (10 points) Thread safety
   
   (a) (5 points) Define what it means for a class to be thread-safe.

   (b) (5 points) Can a thread-safe class contain any public instance fields? Explain.
3. (10 points) Java Memory Model

(a) (5 points) Consider the following (partial) event sequence for a Java program containing two threads, $T_1$ and $T_2$. (Numbers are added for later reference.)

For the event pairs below, circle those for which the first event “happens-before” the second event.

i. 1 and 2

ii. 4 and 8

iii. 7 and 10

iv. 10 and 11

v. 9 and 13
(b) (5 points) In a single-threaded Java application the following statements may be used interchangeably (assume x has type int).

\[
\begin{align*}
    x &= x+x; \\
    x &= 2\times x;
\end{align*}
\]

Can they be used interchangeably in a multi-threaded Java program? Explain.
4. (10 points) Visibility and Publishing.

(a) (5 points) Explain why the assignment of a value to a global variable in one thread may not be visible in another thread.

(b) (5 points) Explain why it is generally poor practice to publish an object inside its constructor in a multi-threaded application.
5. (10 points) Deadlock.

(a) (5 points) Explain the difference between nested-monitor lockout and thread-starvation deadlock.

(b) (5 points) In Java every thread is in one of six possible states, simplified descriptions of which are below.

- **NEW**: Thread has been created but not yet started.
- **RUNNABLE**: Thread is running in a JVM.
- **BLOCKED**: Thread is blocked waiting for a monitor lock.
- **WAITING**: Thread is in a wait set.
- **TIMED_WAITING**: Thread is in a wait set, with an upper time limit specified for this waiting.
- **TERMINATED**: Thread has terminated.

Explain how to use thread-state information to distinguish between circular-wait deadlocks due to improper locking and nested-monitor lockout.
6. (10 points) True / False. Indicate whether each of the following statements is “True” or “False” by writing a “T” or “F” to the left of the statement.

(a) Client-side locking should be used to implement compound actions over a thread-safe class.

(b) The Java Monitor Pattern produces thread-safe classes.

(c) Fail-fast iterators never throw an exception.

(d) Tail-recursive algorithms may be parallelized by creating a task for each call, even when the thread pool contains only a fixed number of threads.

(e) It is always safe to embed an object from a fully synchronized class inside another object that is also fully synchronized.

(f) Race conditions and data races are different terms for the same thing.

(g) All read and write operations on variables of primitive types are atomic.

(h) Daemon threads are automatically terminated in an application once all the application’s user threads have terminated.

(i) Fork/join parallelism relies on the use of stacks to support work stealing.

(j) Latches, barriers and locks are examples of synchronizers.
7. (10 points) Executors.
   (a) (5 points) An executor can be in one of three states: RUNNING, SHUTDOWN, or TERMINATED. Explain the difference between the SHUTDOWN and TERMINATED states.
   
   (b) (5 points) Suppose you are designing a fixed-size thread pool for running independent tasks. The number of CPUs your computer has is 8, the desired utilization is 0.75 for the application, and the wait-to-compute ratio is 9/1. How many worker threads should your pool contain? Show your work.
8. (10 points) Remote Method Invocation.

(a) (5 points) Explain how Java’s RMI implementation uses marshaling and unmarshaling.

(b) (5 points) Explain the steps that a server must go through in order to convert an object matching the Remote interface into one that can be called remotely by clients.
9. (10 points) Non-locking data structures.

(a) (5 points) Suppose we have a class with the following instance field.

```java
private int val;
```

Implement a method called `compareAndSet` that has the same behavior as a compare-and-set atomic operation. Your method should return type `boolean` and take two `int` arguments.

(b) (5 points) One criticism that is sometimes made against lock-free data structures is that they may admit starvation: a thread executing an operation is not necessarily guaranteed to terminate. Consider the following implementation given in class of the `push()` operation for stacks.

```java
AtomicReference<Node<E>> top = newAtomicReference<Node<E>>();

public void push(E item) {
    Node<E> newHead = new Node<E>(item);
    Node<E> oldHead;
    do {
        oldHead = top.get();
        newHead.next = oldHead;
    } while (!top.compareAndSet(oldHead, newHead));
}
```

Explain how starvation can occur for a thread executing a `push()` operation.
10. (10 points) Programming. In executors worker threads execute tasks that are submitted to the executor. The Producer / Consumer pattern is typically used; a shared blocking queue holds the tasks to be executed, and each worker thread waits to take elements from the queue and then executes them. When a worker thread finishes a task it waits again for a new task from the queue.

Complete the implementation started below for a class WorkerThread by implementing the run() method. Your method should repeatedly wait for an element on the work queue, then execute it. If an InterruptedException occurs while the thread is waiting on the queue, it should terminate normally. If the task being executed throws an exception, the run method for simplicity should ignore the exception and return to waiting for the next task.

The following method from BlockingQueue<E> may be useful.

E take() Returns an element from the queue if the queue is non-empty; blocks otherwise. May throw InterruptedException.

```java
public class WorkerThread extends Thread {
    private final BlockingQueue<Runnable> workQueue;

    public WorkerThread (BlockingQueue<Runnable> workQueue) {
        this.workQueue = workQueue;
    }

    @Override
    public void run() { ... }
}
```