1.

Given:

```
1.  public class WaitTest {
2.     public static void main(String [] args) {
3.        System.out.print("1 ");
4.        synchronized(args){
5.           System.out.print("2 ");
6.           try {
7.              args.wait();
8.           }
9.           catch(InterruptedException e){}
10.       }
11.       System.out.print("3 ");
12.    }
13. }
```

What is the result of trying to compile and run this program?

A. It fails to compile because the `IllegalMonitorStateException` of `wait()` is not dealt with in line 7

B. `1 2 3`

C. `1 3`

D. `1 2`

E. At runtime, it throws an `IllegalMonitorStateException` when trying to wait

F. It will fail to compile because it has to be synchronized on the `this` object

Answer:

☑ D is correct. 1 and 2 will be printed, but there will be no return from the `wait` call because no other thread will notify the main thread, so 3 will never be printed. It's frozen at line 7.

☒ **A** is incorrect; `IllegalMonitorStateException` is an unchecked exception. **B** and **C** are incorrect; 3 will never be printed, since this program will wait forever. **E** is incorrect because `IllegalMonitorStateException` will never be thrown because the `wait()` is done on `args` within a block of code synchronized on `args`. **F** is incorrect because any object can be used to synchronize on and `this` and `static` don't mix.

2.

Assume the following method is properly synchronized and called from a thread A on an object B:

```
wait(2000);
```

After calling this method, when will the thread A become a candidate to get another turn at the CPU?

A. After object B is notified, or after two seconds

B. After the lock on B is released, or after two seconds

C. Two seconds after object B is notified

D. Two seconds after lock B is released

Answer:

☑ **A** is correct. Either of the two events will make the thread a candidate for running again.

☒ **B** is incorrect because a waiting thread will not return to runnable when the lock is released, unless a notification occurs. **C** is incorrect because the thread will become a candidate immediately after notification. **D** is also incorrect because a thread will not come out of a waiting pool just because a lock has been released.

3.

Which are true? (Choose all that apply.)

A. The `notifyAll()` method must be called from a synchronized context

B. To call `wait()`, an object must own the lock on the thread

C. The `notify()` method is defined in class `java.lang.Thread`

D. When a thread is waiting as a result of `wait()`, it releases its lock

E. The `notify()` method causes a thread to immediately release its lock

F. The difference between `notify()` and `notifyAll()` is that `notifyAll()` notifies all waiting threads, regardless of the object they're waiting on

Answer:

☑ **A** is correct because `notifyAll()` (and `wait()` and `notify()`) must be called from within a synchronized context. **D** is a correct statement.

☒ **B** is incorrect because to call `wait()`, the thread must own the lock on the object that `wait()` is being invoked on, not the other way around. **C** is wrong because `notify()` is defined in `java.lang.Object`. **E** is wrong because `notify()` will not cause a thread to release its locks. The thread can only release its locks by exiting the synchronized code. **F** is wrong because `notifyAll()` notifies all the threads waiting on a particular locked object, not all threads waiting on *any* object.

4.

Given:

```
public static synchronized void main(String[] args) throws
 InterruptedException {
    Thread t = new Thread();
    t.start();
    System.out.print("X");
    t.wait(10000);
    System.out.print("Y");
}
```

What is the result of this code?

A. It prints X and exits

B. It prints X and never exits

C. It prints XY and exits almost immeditately

D. It prints XY with a 10-second delay between X and Y

E. It prints XY with a 10000-second delay between X and Y

F. The code does not compile

G. An exception is thrown at runtime

Answer:

☑ **G** is correct. The code does not acquire a lock on `t` before calling `t.wait()`, so it throws an `IllegalMonitorStateException`. The method is `synchronized`, but it's not synchronized on `t` so the exception will be thrown. If the wait were placed inside a `synchronized(t)` block, then the answer would have been **D**.

☒ **A, B, C, D, E,** and **F** are incorrect based the logic described above.

5.

Given:

```
public class TwoThreads {
    static Thread laurel, hardy;
    public static void main(String[] args) {
        laurel = new Thread() {
            public void run() {
                System.out.println("A");
                try {
                    hardy.sleep(1000);
                } catch (Exception e) {
                    System.out.println("B");
                }
                System.out.println("C");
            }
        };
        hardy = new Thread() {
            public void run() {
                System.out.println("D");
                try {
                    laurel.wait();
                } catch (Exception e) {
                    System.out.println("E");
                }
                System.out.println("F");
            }
        };
        laurel.start();
        hardy.start();
    }
}
```

Which letters will eventually appear somewhere in the output? (Choose all that apply.)

A. A

B. B

C. C

D. D

E. E

F. F

G. The answer cannot be reliably determined

H. The code does not compile

Answer:

☑ **A, C, D, E,** and **F** are correct. This may look like `laurel` and `hardy` are battling to cause the other to `sleep()` or `wait()`—but that's not the case. Since `sleep()` is a `static` method, it affects the current thread, which is `laurel` (even though the method is invoked using a reference to `hardy`). That's misleading but perfectly legal, and the Thread `laurel` is able to sleep with no exception, printing `A` and `C` (after at least a 1-second delay). Meanwhile `hardy` tries to call `laurel.wait()`—but hardy has not `synchronized` on `laurel`, so calling `laurel.wait()` immediately causes an `IllegalMonitorStateException`, and so `hardy` prints `D`, `E`, and `F`. Although the *order* of the output is somewhat indeterminate (we have no way of knowing whether `A` is printed before `D`, for example) it is guaranteed that `A`, `C`, `D`, `E`, and `F` will all be printed in some order, eventually—so **G** is incorrect.

☒ **B, G,** and **H** are incorrect based on the above.

6.

Consider the following program:

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

class AtomicIntegerTest {
    static AtomicInteger ai = new AtomicInteger(10);
    public static void check() {
            assert (ai.intValue() % 2) == 0;
    }
    public static void increment() {
            ai.incrementAndGet();
    }

    public static void decrement() {
            ai.getAndDecrement();
    }
    public static void compare() {
            ai.compareAndSet(10, 11);
    }
    public static void main(String []args) {
            increment();
            decrement();
            compare();
            check();
            System.out.println(ai);
    }
}
```

The program is invoked as follows:

```
java -ea AtomicIntegerTest
```

What is the expected output of this program?

A. It prints 11.

B. It prints 10.

C. It prints 9.

D. It crashes throwing an `AssertionError`.

Answer:

D. It crashes throwing an `AssertionError`.

(The initial value of `AtomicInteger` is 10. Its value is incremented by 1 after calling `incrementAndGet()`. After that, its value is decremented by 1 after calling `getAndDecrement()`. The method `compareAndSet(10, 11)` checks if the current value is 10, and if so sets the atomic integer variable to value 11. Since the assert statement checks if the atomic integer value `% 2` is zero (that is, checks if it is an even number), the assert fails and the program results in an `AssertionError`.)

7.

Which one of the following options correctly makes use of `Callable` that will compile without any errors?

A.
```
import java.util.concurrent.Callable;

class CallableTask implements Callable {
        public int call() {
                System.out.println("In Callable.call()");
                return 0;
        }
}
```

B.
```
import java.util.concurrent.Callable;

class CallableTask extends Callable {
        public Integer call() {
                System.out.println("In Callable.call()");
                return 0;
        }
}
```

C.
```
import java.util.concurrent.Callable;

class CallableTask implements Callable<Integer> {
        public Integer call() {
                System.out.println("In Callable.call()");
                return 0;
        }
}
```

D.
```
import java.util.concurrent.Callable;

class CallableTask implements Callable<Integer> {
        public void call(Integer i) {
                System.out.println("In Callable.call(i)");
        }
}
```

——————————————————

Answer: C

(The `Callable` interface is defined as follows:

```
public interface Callable<V> {
    V call() throws Exception;
}
```

8.

Which one of the following methods return a `Future` object?

A. The overloaded `replace()` methods declared in the `ConcurrentMap` interface

B. The `newThread()` method declared in the `ThreadFactory` interface

C. The overloaded `submit()` methods declared in the `ExecutorService` interface

D. The `call()` method declared in the `Callable` interface

Answer:

C. The overloaded `submit()` methods declared in `ExecutorService` interface

Option A) The overloaded `replace()` methods declared in the `ConcurrentMap` interface remove an element from the map and return the success status (a Boolean value) or the removed value.

Option B) The `newThread()` is the only method declared in the `ThreadFactory` interface and it returns a `Thread` object as the return value.

Option C) The `ExecutorService` interface has overloaded `submit()` method that takes a task for execution and returns a `Future` representing the pending results of the task.

Option D) The `call()` method declared in `Callable` interface returns the result of the task it executed.)

9.

You're writing an application that generates random numbers in the range 0 to 100. You want to create these random numbers for use in multiple threads as well as in `ForkJoinTasks`. Which one of the following options will you use for less contention (i.e., efficient solution)?

A. `int randomInt = ThreadSafeRandom.current().nextInt(0, 100);`

B. `int randomInt = ThreadLocalRandom.current().nextInt(0, 101);`

C. `int randomInt = new Random(seedInt).nextInt(101);`

D. `int randomInt = new Random().nextInt() % 100;`

Answer:

B. `int randomInt = ThreadLocalRandom.current().nextInt(0, 101);`

(`ThreadLocalRandom` is a random number generator that is specific to a thread. From API documentation of this class: "Use of the `ThreadLocalRandom` rather than shared Random objects in concurrent programs will typically encounter much less overhead and contention."

The method "`int nextInt(int least, int bound)`" in the `ThreadLocalRandom` class returns a pseudo-random number that is uniformly distributed between the given least value and the bound value. Note that the value in parameter `least` is inclusive of that value and the `bound` value is exclusive. So, the call `nextInt(0, 101)` returns pseudo-random integers in the range 0 to 100.)

10.

In your application, there is a producer component that keeps adding new items to a fixed-size queue; the consumer component fetches items from that queue. If the queue is full, the producer has to wait for items to be fetched; if the queue is empty, the consumer has to wait for items to be added.

Which one of the following utilities is suitable for synchronizing the common queue for concurrent use by a producer and consumer?

A. `RecursiveAction`

B. `ForkJoinPool`

C. `Future`

D. `Semaphore`

E. `TimeUnit`

Answer:

D. `Semaphore`

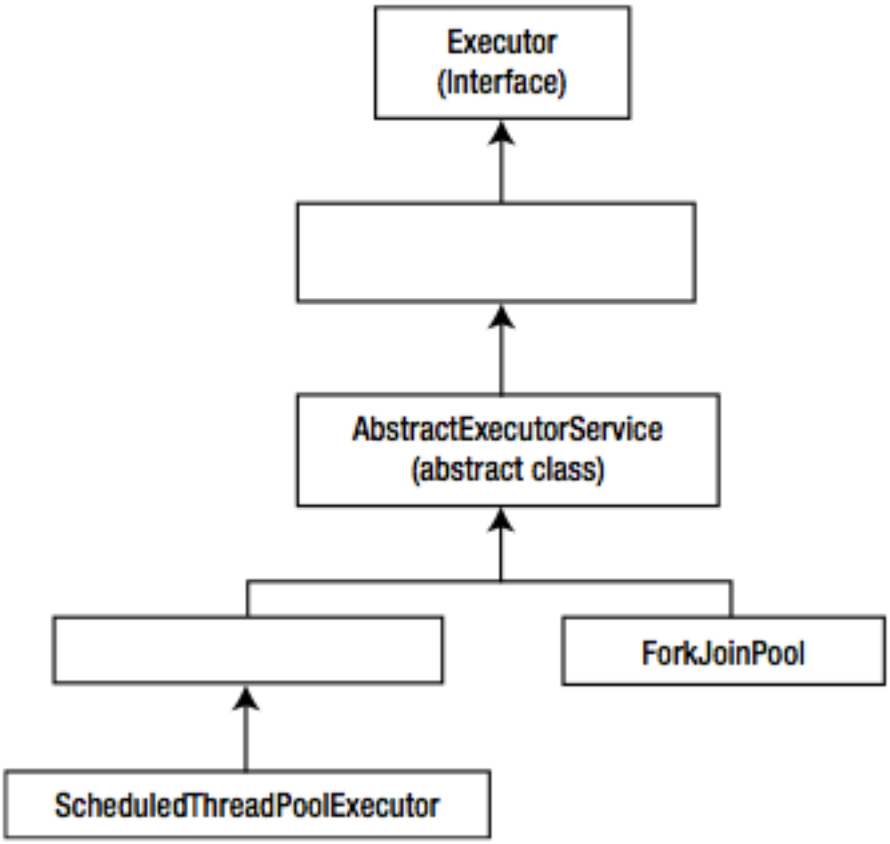
(The question is a classic producer–consumer problem that can be solved by using semaphores. The objects of the synchronizer class `java.util.concurrent.Semaphore` can be used to guard the common queue so that the producer and consumer can synchronize their access to the queue. Of the given options, semaphore is the only *synchronizer*; other options are unrelated to providing synchronized access to a queue.

Option A) `RecursiveAction` supports recursive `ForkJoinTask`, and option B) `ForkJoinPool` provides help in running a `ForkJoinTask` in the context of the Fork/Join framework. Option C) `Future` represents the result of an asynchronous computation whose result will be "available in the future once the computation is complete." Option E) `TimeUnit` is an enumeration that provides support for different time units such as milliseconds, seconds, and days.)

11. Верно ли, что методы isShutdown и isTerminated объявлены в интерфейсе Executor?

Ответ: нет, они объявлены в интерфейсе ExecutorService.

12. Дополните недостающие элементы:



Ответ:
ExecutorService (interface) и ThreadPoolExecutor, соответственно.

13. Какой интерфейс реализуют классы, которые используется при реализации паттерна читатель/писатель в мультипоточном окружении?

Ответ: ReadWriteLock

14. Назовите класс из JDK, который реализует этот интерфейс.

Ответ: ReentrantReadWriteLock

15. Сколько объектов Condition (инстансов классов, реализующих интерфейс java.util.concurrent.locks.Condition) можно запросить у объектов класса ReentrantLock?

A. 1
B. 2
C. сколько угодно
D. по количеству тредов, которые заблокированы данным локом
E. объекты Condition не запрашиваются у ReentrantLock

Ответ: C. сколько угодно