

## 1. (a)

- The access level determines from which class, sub-class, package, etc. the member can be accessed. The access class of a member is declared via one of the following modifiers: [2]
- **public**: member can be accessed from anywhere in the application and hence forms the public interface of a class. [2]
- **protected**: member can be accessed from within the class, its package and all its subclasses and hence allows sub-classes and classes in other packages to have access to implementation details of their superclass. [2]
- **no modifier**: member can only be accessed from within its class and its package and hence is limited to be used within a package, e.g., as a helper (or the program). [2]
- **private**: member can only be accessed from within the class and hence is specific to the implementation of the class only. [2]

Total: [10]

## 1. (b)

- In event-driven programming the flow of the program is determined by events, e.g. for user-interfaces the input of a user determines which task is being executed. An event here is an abstraction for an input to the program that can occur at any time. The abstract notion of an event occurring triggers an action representing the transition from one state of the program to another (i.e. usually a change of the variables describing the state of the program; closely related to finite state machines). [5]
- The object types involved is an event source (publisher) which triggers the event, an object describing the event itself, and an event listener (observer) executing the state transition (action). The listener is registered with the source to receive the event object when the event occurs. [5]
- One way to implement an event processing system is to have a queue storing the events. When an event occurs it will be added to the queue by the source object. A separate process implemented in the event processing system fetches the events from the queue, locates all registered event listeners via the source object, which calls the listeners in sequence (hence they are directly registered with the source and not via the processing system). [5]

Total: [15]

2. (a)

- Extend `Thread` class and implement `run ()` method. [2]
- For cancellation check repeatedly if interruption is set and catch interrupted exception. Thread is interrupted via call to `cancel ()` method. [3]
- ```
public class MyThread extends Thread {
    public MyThread () { ... /* setup */ }
    public void cancel () { interrupt (); }
    public void run () {
        try {
            while (!Thread.currentThread () . isInterrupted () &&
                ... /* more checks */) {
                ... // Execute task
            } }
        catch (InterruptedException consumed) { }
    } }
```

 [5]

Total: [10]

2. (b)

- The mutable shared state of a program must be protected, i.e. groups of variables with invariants/constraints (if more than one variable) that are not constant must only be accessed after a lock has been obtained to ensure atomicity and visibility. (After the access the lock has to be released). To modify as well as to read a state the same lock object for the state must be used. [5]
- Atomicity: an operation on the state is executed completely either before or after any other operation on the same state. Obtaining a lock before the operation (consisting of more than one instruction) and releasing it afterwards ensures atomicity. Otherwise race conditions can occur which means the instructions of multiple atomic operations are executed interleaved which in turn may lead to violations of the invariants of the state (e.g. read-modify-write or check-then-act operations). [5]
- Visibility: accessing the state (variables) without obtaining a lock may mean a stale value is obtained if the variables are accessed by more than one thread due to values being stored in local registers, caches, etc. not accessible from every thread. Without synchronisation only some values may be updated, and the update of variable values does not necessarily occur in the same order they were changed. (Out-of-thin-air safety guaranteed for 32-bit values; volatile variables always visible, but atomicity not guaranteed—can at most be modified on single thread or be constant). [5]

Total: [15]

2. (c)

- The program is very likely to deadlock, i.e. the two threads created will wait for each other indefinitely to obtain the locks on the `n1` and `n2` objects. This situation can always occur when a thread has to obtain more than one lock simultaneously to execute the operation. [3]
- The specific problem is the nested synchronisation blocks. Thread 1 may obtain the lock on `n1`. Then Thread 2 can obtain the lock on `n2`, and then waits for thread 1 to release the lock on `n1` indefinitely as thread 1 tried to obtain the lock on `n2` which is owned by thread 2 and hence waits for thread 1 to complete (or similar situations). [2]
- One strategy to avoid this problem is to enforce to get the locks of the objects always in one specific global order, e.g. for this simple program it

is sufficient to always get the lock on n1 first and then on n2 to avoid the deadlock. [3]

- In general the order may be enforced using a unique number such as Java's object id hash to order the locks. In rare situations where two objects have the same number a tie-breaker lock can be obtained before the locks of the other objects. [2]

Total: [10]

3.

- ```
class Worker extends Runnable {
    private Socket incoming;
    private Executor exec;
    Worker (Socket s, Executor e) { incoming = s; exec = e;}
    public void run () {
        if (new Helper () . task (incoming)) {
            exec.shutdown ();
        }
    }
}
```

[5]

- ```
public class ThreadedExecServer {
    private static final ExecutorService exec =
        Executors.newFixedThreadPool (12);
    public static void main (String[] args) {
        try {
            final ServerSocket serverSocket = new ServerSocket (8080);
            while (!exec.isShutdown ()) {
                Socket incoming = serverSocket.accept ();
                exec.execute (new Worker (incoming));
            }
        } catch (Exception e) {
            if (!exec.isShutdown ()) {
                System.out.println (e);
            }
        }
    }
}
```

(or variations of this)

[10]

- If more requests than threads are arriving they will be queued in the executor service until a thread becomes available. Response to individual requests may be delayed, but running requests are not slowing down too much. However, there is a risk of too many requests arriving such that the resources (memory) of the executor service are exhausted. In this case the server will crash, instead of denying more requests (details of answer depend on executor service used).

[5]

Total: [20]