14.

1) The code is not thread safe because modifications on one thread may be lost due to modifications on another thread. It's the `a += m` line that's not safe. **[1]**

If two threads were running concurrently and calling the `increaseA` method, then some of the updates could potentially be lost; `a`'s value will not be the sum of all of the updates applied from all of the threads using the object. **[1]**

2) If thread A performs two actions, *x* and *y*, then the actions are done in that order from thread A's viewpoint. However, another thread may observe action *y* being performed before action *x*. The purpose of `volatile` forces the actions taken in one thread to appear in another thread in the same order. **[1]**

Making `a volatile`, does not change the order of actions involved in `a += m`; therefore, the code is semantically the same, hence there can't be an improvement in thread safety. Another approach to this answer is just to state that the only action that `volatile` makes atomic is assignment on long and double (32bit assignments are already considered atomic). It does not make an operation such as `+=` atomic, not even on ints.

3) There are several ways this can be approached, including the two non solutions mentioned previously. The differences in suitability and performance are listed in the table below:

Technique	Execution Time	% Lost Updates
No Protection	0.023	49%
Volatile Keyword	9.062	25%
Synch Method	21.301	0%
Synch Object	21.492	0%
Atomic Integer	3.193	0%

The code used to generate the table created a single `A` object and ran two threads concurrently each calling `increaseA` with an argument of 1. Each thread called the method 100,000,000 times. Tests were run on a dual processor (important!) machine.

Any solution without protection, or using just `volatile` is just plain wrong.

The remaining solutions are all reasonable but some are better than the others. The synch method is the easiest to implement and is done by attaching the `synchronized` keyword onto the `increaseA` method. **[1 mark for this solution]**

Synch object creates an internal object to use a lock and hence the `synchronized` keyword is used inside the method. This is better because of increased encapsulation and doesn't add to the complexity of the interface **[1.5 marks for this solution]**

`AtomicInteger` is the best choice here. Interestingly it performed slightly faster than when the `volatile` keyword was used. **[2 marks for this solution]** Example solution:

```
class A
{

	private AtomicInteger a = new AtomicInteger();

	public int getA() {
		return a.get();
	}

	public void increaseA(int m){
		a.addAndGet(m);
	}
}
```

Any of the other reasonable solutions should net one mark **[1]** and any reasonable answer should be accepted, e.g. suggesting that the `synchronized` keyword on the method is a well understood method of synchronizing is a justifiable reason if they chose this method instead of `AtomicInteger` **[1]**.

4) `AtomicInteger` is no longer a suitable mechanism because now the class will need to maintain an invocation count variable of so[illegible]kind. Because there are two pieces of data, we need a lock that can be applied over both. This can now be done using either a `synchronized` method, or using an internal locking object (as suggested as reasonable solutions above) **[1]**. If the answer provided for (3) uses an `AtomicInte[illegible]` expected, then the fact that an `AtomicInteger` merely makes the actions on a single int a[illegible]mic me[illegible]s that it is not suitable for protecting more than one variable. **[1]**

18.

We want the loaded images to stay in memory while memory is available, but if memory is tight, then they should be discarded. We can do this using one of the other types of reference **[1]**. Out of the three types available (soft, weak, and phantom), soft reference is the best choice **[1]**. Softly reachable objects are removed when the JVM thinks it's running out of memory, therefore the images will remain in memory for as long as possible **[1]**. Other solutions don't take advantage of the JVM's understanding of memory, and other references won't be cleared at the best time (e.g. weak is always cleared at garbage collection) **[1]**.

There are other schemes that could be considered and any reasonable scheme should be awarded some points. However, it is doubtful that any other scheme would have the simplicity of using soft references.

To make this code as easy to use as possible, you would create a proxy object that when created loaded the image and maintained a soft reference to it. When the proxy is requested to do something, it would check the soft reference to see whether it was valid. If the soft reference was not valid (null), then it would reload the resource. **[1]**

In order to provide a good design, the proxy object must look like an image object, that means it should support the original interfaces (or derive from) the image that is currently being used.**[1]** This could possibly mean deriving your proxy from a class such as `BufferedImage`.