# Santa Klaus Workshop Plan

## I. Oh, Christmas joy!

Christmas is just around the corner! Santa Klaus and his staff (elves and reindeers) are preparing for another wonderful Christmas that brings joy and gifts to all children around the world.

***But Santa Klaus is in need for some help!***

He owns a workshop that may contain multiple factories (every year he recreates the factories). He has elves in the factories creating toys, he has reindeers awaiting the toys to wrap them up in gifts and he desperately needs a workshop running (because his older workshop manager left). The workshop plan has some rules that will ensure that all gifts are created in due-time and that no child is left without a present (would be such a pity).

He found out from us that you are all ready to help him and implement a factory plan by using concurrency in Java (he knows already you are all experts in the field) that will provide the needed help.

*(Happy Santa about the news that he gets help)*

Now below you can find some information that we know and found out from Santa about the workshop and things that are needed to be done:

## II. Things that Santa told us

He instructed us to tell you the following things that he has in his workshop:

1. He has **multiple** toys factories:


*(Toy Factory, such a wonderful building)*

- Random number of factories (between 2-5) that create toys (he doesn't know how many are needed until your factory plan is created and runs)
- All factories are in matrix like form (random NxN, where N is between 100-500)
- Each factory contains an array list with elves
- Every few seconds the factory will ask all its elves their position in the factory
- There can be no more than N/2 number of elves in a factory

2. He has **elves**. Elves are really important; they create all the nice toys. Here's what he told us about them:

*(Happy elf working hard)*

- Elves are spawning randomly in each factory at a random place in the factory (random time between 500-1000 milliseconds)
- When elves are spawned randomly they need to tell the factory that they are there
- No two elves can be on the same position
- Each elf works independently of any other elf
- Elves create gifts by:

- Moving in one direction (left, right, up, down)
- When they reach a wall of the factory they move in any other direction than the wall's
- Once they move they also create the gift
- If an elf is surrounded by other elves and cannot move it stops for a random time (between 10-50 milliseconds)
- When the gift is created they update their factory to know what gift they created and the factory will ask all elves to pass on the information on their location to give it to Santa
- After the elf creates a gift, he needs to rest for a short while (30 milliseconds)
  *) They are really fast
- Elves will work like in… forever (well, at least until December 25th. As such they don't need to stop.

3. He also has **reindeers**:

*(Awaiting reindeer, wondering if he receives all gifts to pass to Santa)*

- Reindeers await to receive gifts from the factory. Reindeers are available in the workshop
- They can always read from the factories the number of gifts (and which gifts), but they won't be allowed access when elves notify the factories about new gifts or when the factory itself asks the elves about new gifts
- A maximum of 10 reindeers can read from a factory at a time
- A random time must pass between two consecutive factory readings
- Reindeers will put all gifts through a pipe to give to Santa
- Multiple reindeers at the same time put the gifts while Santa reads by itself all gifts
- Reindeers are known from the beginning (there are more than 8, because Santa needs backup).

4. The main Santa's Workshop will do the following:


*(Sign to Santa's workshop hidden at North Pole)*

- this workshop contains all factories
- it creates the factories
- it also spawns elves and gives them a random factory to work in (remember that each elf is responsible to register himself to the factory)

## III. Things that are necessary

Please note that the (concurrent) control flow in this example is rather subtle. The moving of elves and reporting to the factory is initiated independently by each of the elves, as each of them is running a separate thread of control.

Because of that, different elves may execute the move action and factory reporting concurrently. Concurrent factory reporting has to also be synchronized.

But calling report also triggers calling the individual report method of all the elves registered with the factory. This again has to be properly taken care of, because both report and move methods are synchronized, which means that they will not be called concurrently for a given object.

## IV. Some hints

Santa knows you can help him. He knows that you have in your tool belt the following concurrent notions that you can help him with:
- Semaphores, Monitors and Locks (he knows that you will fairly use all of them)
- Threads are really important for him. He even said that each of his elf can be a separate thread in your factory plan implementation
- He wants them to communicate: the elves create the gifts and the reindeers receive them and wrap them up in nice packages to give to Santa to deliver
- And Santa dearly wants the reindeers to deliver him the gifts to his office through TCP/IP (he found out this is good). If he receives all gifts then he can create the list for the children (don't miss any gift)

## V. Extra tasks

### *1. You can retire an elf*

Your first task is to add still another thread to your program that retires the elves at random times (i.e. with short random delays in between). But the elves should be retired in random order. Retiring an elf means that its run method must terminate. This should be achieved by making the loop terminate normally and NOT by calling the deprecated method to stop a thread. Further, the elf must be removed from the factory (in a thread-safe way similar to adding an elf to a factory). After this has been done the garbage collector in the run-time system will eventually reclaim the space allocated for the elf object.

To solve the problem you can make use of a semaphore that the elves try to acquire in order to “get permission to retire”. The semaphore is initially zero and then released a number of times in a thread started in main. Note that it is very useful to try to acquire the semaphore, using its tryAcquire method.

Implement this and test your program. Make sure that you understand how the design makes the retiring order unpredictable. If you want, you can change the behavior of the program further so that retired elves are respawn after some (random) time.

### *2. You can give an elf time to sleep (because you've noticed that he/she is really tired)*

Now return to the original version of the program (make a new directory and reuse the initial set of classes you implemented).

You must now modify the program to achieve the following behavior: When an elf after one of its moves finds itself in the diagonal area of the world (i.e. where x is very close to y), it will "go to sleep", i.e. stop moving. Note that an elf may jump over the diagonal area in one move; this will not cause it to sleep. When all elves have frozen at the diagonal, they will all wake up and continue moving until they sleep again on the diagonal. This moving/sleeping continues forever.

You should recognize this as a form of barrier synchronization that can be achieved using N + 1 semaphores: one common barrier semaphore, which elf threads release when they reach the synchronization point, and an array of "continue" semaphores, indexed by thread, which threads acquire in order to continue beyond the barrier.

A special barrier synchronization process is also needed, which repeatedly acquire the barrier semaphore N times, followed by releasing all the continue semaphores.

### *3. Sleeping elves revisited*

The package java.util.concurrent includes the class CyclicBarrier, which provides more convenient means to achieve barrier synchronization. Rewrite the program from the previous exercise using this class instead of semaphores.

### *4. Your own cycle barrier*

Now for a harder exercise: implement the body of your own class CyclicBarrier which provides similar features as the Java class of the same name. But we are content with a simpler version with the following spec:

```
public class CyclicBarrier {

   public CyclicBarrier(int parties);

   public void await();
}
```

The parameter parties are the number of threads that need to reach the barrier before they are all allowed to continue. Write the whole class and then use it to solve the sleeping elf problem again.

Hint: We cannot follow the array-of-semaphores approach directly, since that would require await to have a parameter *i* indicating the index of the semaphore on which to block. Instead, one could try to use a single semaphore on which all processes block and an integer variable counting how many

processes have reached the barrier. But this integer variable is shared, so we need to protect updates to it using a second, mutex semaphore. We cannot use the Java synchronized locking instead of the mutex semaphore. Why?