PROJECT DESCRIPTION AND DOCUMENTATION

1.CLASS ChristmasGift

Implementation of a Christmas gift under the form of a class; this is a simple object structure that is needed in the micro world of the program in order to simulate a North Pole Workshop

private giftIndex: index of the Christmas gift

private isPacked: boolean member; it tells whether the gift has been packed and is set to true after being processed by a reindeer

private numberOfGifts: Static member that is an internal class counter; it gives the index for the gift (each gift has a number starting from 0; each time we create a gift, it gets an index greater by one from the previous one; initialised when an object is firstly created, trough the use of a static initializer

public ChristmasGift(boolean isPacked): Constructor for gift sets the isPacked property, the gift index from the static member numberOfGifts and increments the latter member

public toString(): method from the Object class has been overridden in order for it to return, after constructing it with a StringBuilder, a message; the message tells us the number of the gift, and whether it is unpacked or if it has already been packed by a reindeer

2.Class ConcurrentQueue<T>

A particular implementation of a first in / first out (FIFO) queue that works correctly in a situation in which multiple threads try to access it at the same time, either for adding new values or for removing already added ones Two methods have been implemented, the enq(), for adding a new value at the end (tail) of the queue and the deq(), for removing the first element (head) of the queue This is a generic class, meaning that the "T" value can be replaced with an Integer, String, Character, RandomCharacter (and so on) instance

private volatile int head: The current head of our concurrent queue The volatile keyword guarantees that any thread that reads this member will see the most recently written value

private volatile int tail: The current tail of our concurrent queue The volatile keyword guarantees that any thread that reads this member will see the most recently written value

private T[] queueValues: The generic array that is the actual concurrent queue in which all operations will be performed (values added and then consumed)

ReentrantLock queueLock: An instance of the ReentrantLock -implementation of the Lock interface-, the queueLock is fundamental in ensuring the synchronization of the queue operations performed by the producers and consumers threads

Condition queueCondition: A condition variable belonging to the reentrant lock, used in order to perform await(), signal(), signalAll and so on, operations used in order to ensure that we manage with the cases in which the queue is either empty or full

private int queueMaxSize: A member which holds the maximum length of the queue, so the number of elements that can fit inside the queue until it is full `public ConcurrentQueue(int queueMaxSize)`: The constructor gets the maximum dimension of the queue; it initialises the head and the tail as being 0, so no element is present in the queue in this moment It creates a new reentrant lock instance and uses its newCondition() method in order to obtain a condition variable; we create a generic queue that will hold a maximum number of queueMaxSize elements; created as an array of Object instances, it is then casted to the generic type T

`public void pushIntoConcurrentQueue(T newValue)`: A method that adds a new value to the end of the queue At first, it tries to acquire the lock; if it succeeds, it tries the following: if the queue is full and no elements can be added at the moment, it awaits in a while loop; once it gets past the while loop (there is now room to add a new value), it calculates the position in which the new element must be inserted and then adds it, increasing afterwards the value of the tail so that it indicates now to the next position in which a value can be inserted A message is printed on the console, indicating the value that has been added Then, as the queue has been modified, it notifies all other processes so that they will get their chance at performing an action on the concurrent queue In a finally block, we call the unlock() method so we will be sure that, whatever happens in the try block, the lock will be available for other processes to take and use it, avoiding deadlock

`public T popFromConcurrentQueue()`: A method that removes a value from the head of the queue At first, it tries to acquire the lock; if it succeeds, it tries the following: if the queue is empty and no elements can be removed at the moment, it awaits in a while loop; once it gets past the while loop (there is at least a value in the queue), it calculates the position of the first element (the head index) and then removes it form the queue, increasing afterwards the value of the head so that it indicates now to the next position in which the first valid element can be found A message is printed on the console, indicating the value that has been removed and consumed Then, as the queue has been modified, it notifies all other processes so that they will get their chance at performing an action on the concurrent queue In a finally block, we call the unlock() method so we will be sure that, whatever happens in the try block, the lock will be available for other processes to take and use it, avoiding deadlock

`private boolean isQueueFull()`: A simple boolean method that checks if the queue is full, by verifying if the difference between the tail and the head is equal with the maximum capacity of our queue; if it is not the case, the method returns false

`private boolean isQueueEmpty()`: A simple boolean method that checks if the queue is empty, by verifying if the tail is equal with the head (meaining that there are no elements left) if it is not the case, the method returns false

3.Class ElfStateGetter

Service Thread called ElfStateGetter that has a very clear task: it performs a number of queries at random intervals (1, 2 seconds) trough which he asks elves from all factories about their current position; the method that does this is synchronized by using a binary semaphore because the reindeer must not be able to get presents from the factories when they are being questioned by the ElfStateGetter; so reindeer must firstly acquire the semaphore in order to have access to the factory concurrent queue; and there they will have to acquire the queue lock in order not to perform an extraction in the same moment in which an elf is pushing a newly created gift into it; in this way, reindeer cannot get gifts when either the elves are pushing new ones in the queue or when the service thread is querying the elves trough the toy factories

private List<ToyFactory> toyFactoryList: List of all toy factories

private Semaphore serviceThreadSemaphore: Service thread semaphore that coordinates the relation between the elf state getter and the reindeer

private boolean hasAcquiredSemaphore: Boolean member which tells us if the semaphore has been acquired in order to release it; I do this because putting the semaphore in a try finally like in the case of locks is not okay because, a semaphore.acquire can throw and Exception and, in that case, if we call semaphore.release in the finally block, the semaphore will get a bonus permit; because a permit has not been given so we must not try to release; that is why, if the acquire succeeds, I make the member true and, after finishing the job, I test it in order to see if a release is necessary

public ElfStateGetter(List<ToyFactory> toyFactoryList, Semaphore serviceThreadSemaphore): Constructor gets as parameters a list of toy factories and the semaphore share by both the service thread and by the reindeer; the corresponding values are set and the hasAcquiredSemaphore gets the value of false `public void run()`: method from the Thread class is overridden in order to achieve the proper behavior needed in our implementation; the thread runs for a minute, because this is the maximum time allocated for the run of the whole program (or until all gifts are created); all this method does is call the private method askElvesAboutTheirPosition() for each toy factory in a while loop

`private void askElvesAboutTheirPosition()`:Private method that performs the main action of the elf state getter thread: gets the status of the elves from each toy factory; first of all, it tries to acquires a permit from the semaphore; if it succeeds, it sets the hasAcquiredSemaphore flag to true and moves on to the next phase in which, for each factory in the list of toy factories, it gets the list of currently working elves and, for each of them, it prints the current state; afterwards, the method sleeps for a random time period between one and two seconds); in the end, the value of the hasAcquiredSemaphore is tested and, in the case in which the semaphore has been acquired, it is released and the flag member is set to false; otherwise, no action is needed

4.Class ElfWorker

Implementation of an elf worker as a thread; elves are plced on a grdi and belong to one toy factory only; they work until the day of Christmas and produce gifts by moving in one direction; in the case in which they are surrounded, they sleep for a bit and, after moving they also rest, but in the rest of the time they just move all over the map; their number will increase until it reaches the order of the grid matrix divided by 2

`private int elfWorkerIndex`: Elf's assigned factory; they know the place in which they work after being assigned to it

`private ToyFactory assignedFactory`: Elf's assigned factory; they know the place in which they work after being assigned to it

`private int numberOfToysProduced`: Every elf will count the number of toys it produced

`private Position2D currentPosition`: Current position in which this elf can be found on the grid matrix

`private boolean runningFlag`: Flag that is true as long as the elf must go on with its work; when made false, the elf stops its work

`public ElfWorker(int elfWorkerIndex, ToyFactory assignedFactory, Position2D currentPosition)`: Constructor that gets the assigned factory and the current position; number of toys produced is made 0 and the running flag is true

`public void run()`: method from the Thread class is overridden; in order not to force him too much, if after one minute the elf is still working, he will be stopped; or, if it is a hard and quick worker, after he has produced a number of 100 gifts; in the run method, we just move it in one direction and check if the number of gifts has reached the target value

`public void restAfterCreatingGift()`: Method that rests an elf for 30 milliseconds after creating a gift `public void sleepWhenSurrounded()`: Method that ensures that, if surrounded, an elf will wait for some time (random between 10 and 50 milliseconds) and then try again to move and create gifts

`public void incrementNumberOfToysProduced()`: Method that increments the number of toys produced by the current elf

`public void notifyFactoryOfSpawnPosition()`: Method that notifies the assigned factory about the fact that the current elf has produced a toy; the value in the grid corresponding with the elf's new position is properly set to true; method is synchronized so that only one elf can notify factory at a time after creation

`public void notifyFactoryOfMoveUp()`: Method that notifies factory after a certain type of move, in this case a move to the north; the current elf position is saved and marked with false in the factory grid because the elf is about to move from that point; using a method from Position2D, a new Position2D object, pointing to the position in the north is generated; this new position is set to true in the grid because the elf will stay there now; finally, the currentPosition field in the elf class is updated

`public void notifyFactoryOfMoveDown()`: Method that notifies factory after a certain type of move, in this case a move to the south; the current elf position is saved and marked with false in the factory grid because the elf is about to move from that point; using a method from Position2D, a new Position2D object, pointing to the position in the south is generated; this new position is set to true in the grid because the elf will stay there now; finally, the currentPosition field in the elf class is updated

`public void notifyFactoryOfMoveLeft()`: Method that notifies factory after a certain type of move, in this case a move to the west; the current elf position is saved and marked with false in the factory grid because the elf is about to move from that point; using a method from Position2D, a new Position2D object, pointing to the position in the west is generated; this new position is set to true in the grid because the elf will stay there now; finally, the currentPosition field in the elf class is updated public void notifyFactoryOfMoveRight(): Method that notifies factory after a certain type of move, in this case a move to the east; the current elf position is saved and marked with false in the factory grid because the elf is about to move from that point; using a method from Position2D, a new Position2D object, pointing to the position in the east is generated; this new position is set to true in the grid because the elf will stay there now; finally, the currentPosition field in the elf class is updated

public toString(): toString() method from the Object class is overridden in order to return the index of the factory where the elf is working, the number of the elf and the current position in which he resides

5.Class NorthPoleWorkshop

Implementation of a Santa Claus Workshop located on the North Pole as a thread; this is the point in our program in which all other threads are started; it can be called a control point or a point of orchestration; here is the place where the christmasGiftPipe shared by Santa Claus and the reindeer is created, the place in which all Reindeer, SantaClaus, Toy Factories and the service threads are started, the point in which the lists of factories and reindeer are located; we also have here the instance of the Semaphore used for the synchronization of the Reindeer and ElfStateGetter thread

`private static final int MAX_GIFT_PIPE_SIZE`: Constant that holds the maximum size of the gift pipe

`private List<Reindeer> listOfReindeer`: List of all reindeer that take gifts from the toy factories and send them trough the pipe to Santa Claus

`private List<ToyFactory> listOfToyFactories`: List of all toy factories in the application

`private int numberOfReindeer`: Member that holds the total number of reindeer

`private int numberOfToyFactories`: Member that holds the total number of toy factories

`private ConcurrentQueue<ChristmasGift> christmasGiftPipe`: Concurrent queue that plays the role of the christmas gift pipe that represents the mean of communication between the reindeer and Santa Claus

`private SantaClaus santaClaus`: Instance of the Santa Claus object

`private ElfStateGetter elfStateGetter`: Instance of the elfStateGetter service thread which gets the position of elves trough factories at certain moments

`private Semaphore serviceThreadSemaphore`: Semaphore instance used in order to synchronize the elfStateGetter thread and the reindeer because they are not allowed to get gifts when the service thread performs a query

`public NorthPoleWorkshop()`: Constructor of the NorthPoleWorkshop takes no parameters; it creates a new fair binary semaphore, generates a random number (between 2 and 5) of toy factories, sets the total number of reindeer as the number of factories multiplied by 10 (we suppose that there are 10 reindeer on each toy factory); it also creates the lists of reindeer and of factories and the concurrent queue that will represent the communication pipe between Santa Claus and the reindeer; then we generate a new Santa Claus instance, a new elfStateGetter, and use another two methods to create all toy factories and all reindeer and store them in the appropriate lists

`public void run()`: method from the Thread class has been overridden in order to obtain the requested functionality; In this method, we start all toy factories which will, on their turn, start to create elves and add them to the grids to work; this is the place in which all reindeer and Santa Claus are also started;

`private ToyFactory generateToyFactory(int toyFactoryIndex)`: Method that generates a toy factory with a given index and a random grid size

`private void createAllToyFactories()`: Method that generates all toy factories by using repetitive calls to the other method, generateToyFactory(int toyFactoryIndex); it creates a random number of toy factories (between 2 and 5) because the member numberOfToyFactories has been randomly generated in the constructor of the class; then it adds all these factories in a list represented by a class member

`private Reindeer generateReindeer(int reindeerIndex, ToyFactory assignedFactory)`: Method that generates a reindeer with a given index and an assigned factory to communicate with in order to get gifts and to pack them

`private void createAllReindeer()`:Method that generates all reindeer by using repetitive calls to the other method, generateReindeer(int reindeerIndex, ToyFactory assignedFactory); it creates sets of reindeer (the number of sets is equal with the number of factories) and in set there are 10 reindeer that will all communicate with an assigned factory; finally, all factories are added in the list represented by a class member

`private int calculateTotalNumberOfGifts()`: Method that calculates, based on a few suppositions, the total number of gifts that can be produced in the factories under the current setup; each toy factory has a maximum number of elves that is equal with half of its grid size and each elf is set to produce at most 100 gifts; the result is returned and represents the number of gifts that must get to Santa Claus

`private SantaClaus generateNewSantaClausInstance()`: Method that generates a new instance of the class Santa Claus; Santa Claus must get the number of kids (calculated trough the use of the private method calculateTotalNumberOfGifts() as each kid will get one gift) and the concurrent queue trough which he communicates with the reindeer; a new Santa Claus with these properties is returned

6.Class Position2D

Implementation of a Position2D pair under the form of a class; This is a simple object structure that is needed in the micro world of the program in order to simulate a North Pole Workshop; it contains two coordinates, the a position on a X axis and a position on a Y axis; it also has getters and setters for those and a few utility methods for handling positions and generating new ones with certain properties

private int positionX: Coordinate on the X axis

private int positionY: Coordinate on the Y axis

public Position2D(int positionX, int positionY): Constructor for the Position2D class that takes as parameters the X and Y coordinates and sets the members accordingly

public Position2D getPositionUp(): Method that generates a new Position2D object starting from the current form of the current one; it gets the position in south of the current one by subtracting one from the Y coordinate finally, it returns a new Position2D object containing the modified values

public Position2D getPositionDown(): Method that generates a new Position2D object starting from the current form of the current one; it gets the position in south of the current one by subtracting one from the Y coordinate finally, it returns a new Position2D object containing the modified values

public Position2D getPositionLeft(): Method that generates a new Position2D object starting from the current form of the current one; it gets the position in west of the current one by subtracting one from the X coordinate finally, it returns a new Position2D object containing the modified values

public Position2D getPositionRight(): Method that generates a new Position2D object starting from the current form of the current one; it gets the position in east of the current one by adding one to the X coordinate finally, it returns a new Position2D object containing the modified values public boolean isPositionValidInGrid(SimpleBooleanGrid testForGrid): Method that checks if a given position is valid in the given grid by checking if both X and Y coordinates are less than or equat to the grid size - 1 and if the coordinates are positive (grid has positions from 0 to grid size - 1)

public boolean isPositionTrueInGrid(SimpleBooleanGrid testForGrid): Method that checks whether a certain position has a value of true in a given SimpleBooleanGrid

public boolean isPositionFalseInGrid(SimpleBooleanGrid testForGrid): Method that checks whether a certain position has a value of false in a given SimpleBooleanGrid

public String toString(): toString() method has been overridden in order for it to return the position object formatted in a easy to understand way

7.Class Reindeer

Implementation of a reindeer under the form of a thread; the reindeer has access to a certain factory alongside with another nine of his kind (I have chosen to put a number of ten reindeer per toy factory in order not to over complicate things up; the reindeer waits at the factory in order to get the gifts that have been produced by elves; it has the task to pack them and then put them in a pipe that goes all the way to Santa's workplace; they are not allowed to get gifts when an elf is putting them in the list of the factory and also when the factory asks its elves about their positions trough a service thread

`private int reindeerIndex`: Current gift being processed by the reindeer; this can be null if the reindeer has not yet taken a gift from the factory in which he works

`private ToyFactory assignedFactory`: Factory in which the reindeer works at taking and packing presents

`private ConcurrentQueue<ChristmasGift> christmasGiftPipe`: Pipe that goes between the reindeer and Santa Claus; all reindeer put their packed gifts in it, but Santa Claus has to take them one by one and add them in his bag

`private ConcurrentQueue<ChristmasGift> factoryGiftQueue`: Queue in which presents are stored at the factory level; the reindeer takes raw presents from it and then packs them

`private boolean runningFlag`: Flag that is set to true in the constructor; as long as it is true, the reindeer will continue to do its job; it is set to false after one minute or when the reindeer packed his part from the total number of produced gifts

`private Semaphore serviceThreadSemaphore`: Binary semaphore shared by the serviceThread that asks elves from all factories about their position and by all reindeer; it ensures that reindeer do not take gifts from the factories when these are giving information about their elves to the service thread (the elf state getter thread) private boolean hasAcquiredSemaphore: Boolean value that is set to false when the reindeer must wait for another reindeer to finish putting a gift in the queue or for the service thread to finish its asking job; it is true when the reindeer has acquired the semaphore

public Reindeer(int reindeerIndex, ToyFactory assignedFactory, ConcurrentQueue<ChristmasGift> christmasGiftPipe,Semaphore serviceThreadSemaphore): Constructor of Reindeer; setting the reindeer index, the toy factory in which he will work, the pipe that goes to Santa and the Semaphore shared with the service thread; the current gift is set to null, the running flag is true and the boolean for the semaphore is false because the reindeer did not yet acquire it

public void run(): method from Thread has been overridden in order to have a behavior for the reindeer thread; we use a private method to calculate the maximum number of gifts that can be produced in the assigned toy factory in order to set a fair share of the work for the reindeer; this is based on the assumption that no more than N / 2 elves can reside in a factory (where N is the order of the matrix) and on the decided number of 10 reindeer that work on a factory; the reindeer works until a minute passes or until it finished its part of work; the process is simple: he takes a gift from the factory by acquiring the binary semaphore (the concurrentQueue ensures that no elf is putting a gift in the queue too), he packs the gift, puts it in the pipe towards Santa Claus and rests for a short while; if it has finished its assigned number of gifts to process, he sets the runnable to false and finishes its work

public String toString(): toString() method has been overridden in order to return the number of the reindeer and the toy factory from which it takes gifts in order to pack them and send them to Santa

private void takeGiftFromFactory(): Method trough which the reindeer tries to take a gift from the factory queue; its first task is to try to acquire the semaphore; itv waits to get it and then it pops a gift from the toy factory queue (it is a concurrent queue so it is ensured that at that time, no elf is trying to put a gift inside of it); finally, if he has finished packing the gift, it releases the semaphore `private void delayBetweenGiftTakingActions()`: Simple method that performs a sleep on the reindeer thread for a random amount of time between 20 and 50 milliseconds

`private void packChristmasGift()`: Method that ensures the packing process that is represented by a call to a ChristmasGift setter that marks the gift as being packed; it also prints to the screen the gift when it arrives unpacked and then when it leaves packed

`private void putGiftIntoPipe()`: Method that puts the packed gift in the pipe; if the reindeer tries to put an unpacked gif, an error message is displayed, but this does not happen in the current problem state because the reindeer firstly packs the gift and then sends it

8.Class SantaClaus

Implementation of Santa Claus under the form of a thread; Santa's job is to get the gifts from the pipe that he shares with all his reindeer; then he has to put it in his bag of presents

private List<ChristmasGift> santaClausBagOfGifts: List that simulates Santa's Christmas bag

private ConcurrentQueue<ChristmasGift> christmasGiftPipe: Pipe that goes from the reindeer squad to Santa's place; all reindeer put their packed gifts in it, and on his side, Santa is taking them alone, one by one, and putting them in his big Christmas bag

private ChristmasGift currentGiftBeingProcessed: Member that saves the current gift that is processed by Santa; it can be null if Santa Claus has not gotten one yet from the pipe

private int numberOfKids: An approximation/estimation of the number of kids for this problem

private boolean runningFlag: Flag that ensures the time length of Santa's work; when true, the thread keeps running; once set to false, Santa finished its job

public SantaClaus(int numberOfKids, ConcurrentQueue<ChristmasGift> christmasGiftPipe): Constructor for the class that sets the number of kids and the shared pipe between Santa Claus and his reindeer; the running flag is set by default to true

public void run(): method from Thread is overridden in order to ensure that Santa does its work correctly; the thread runs for a minute or until all gifts have been processed and put in the Christmas bag; the method consists of a series of method calls: taking the gift from the pipe, putting the current gift in the Christmas bag and finally resting for a short while before getting the next one; in the end, it is checked if the necessary number of toys has been obtained, case in which the thread will finish running public String toString(): toString() method from Object has been overridden in order to return the number of gifts that Santa has in his bag

private void takeGiftFromPipe(): Method that pops a gift from the pipe and stores it in the current gift member

private void putCurrentGiftInChristmasBag(): Method that puts the current gift that has been processed in the Chrismas bag; the method has to be called after processing at least a gift from reindeer; otherwise, an error message is printed

private void restAfterProcessingGift(): Method that ensures the resting period for Santa after the processing process and the addition of a new gift to the bag; random short time between 10 and 20 milliseconds

9.Class SimpleBooleanGrid

Implementation of a Simple boolean grid structure under the form of a class; This is a simple object structure that is needed in the micro world of the program in order to simulate a North Pole Workshop; it wraps a boolean matrix that must play the role of the place in which elves reside and do their work by moving and creating gifts

`private boolean[][] gridMatrix`: Boolean matrix that has a value of true in the position [x][y] if there is currently an elf in the position (x, y); false otherwise

`private int length`: The length of the grid matrix

`private int height`: The height of the grid matrix

`public SimpleBooleanGrid(int orderOfMatrix)`: Constructor gets as a parameter the order of the matrix; this is the one that will be used in the problem, as all matrices are square ones with of a given order; it sets the length and height to be both equal to the order of the matrix and a new background matrix structure is created and filled with false (in the case of the problem, no elves in in at the moment)

`public SimpleBooleanGrid(int length, int height)`: Constructor gets as a parameter the length and height dimensions of the matrix; it sets the length and height accordingly and creates the matrix structure it will be filled with false

`private void fillMatrixWithDefaultValue()`:Private method that fills the matrix with a value of false; it makes use of an enhanced for loop and a method from the Arrays class that resides in the java.util package

`public String toString()`: toString() method from the Object class has been overridden in order for it to return the length and height of the boolean grid

10.Class SimpleGrid<T>

Implementation of a SimpleGrid template structure under the form of a class; This was the basic form for the grid that I intended to use, but I decided to stick with a simpler implementation (left this here as it is a way better way of working with a grid and, I might use it in a future rework); it wraps a template matrix that must play the role of the place in which elves reside and do their work by moving and creating gifts

private T[][] gridMatrix: Template underlying matrix that represents the actual grid

private int length: The length of the grid matrix

private int height: The height of the grid matrix

public SimpleGrid(int orderOfMatrix): Constructor gets as a parameter the order of the matrix; this is the one that will be used in the problem, as all matrices are square ones with of a given order; it sets the length and height to be both equal to the order of the matrix and a new background matrix structure is created

public SimpleGrid(int length, int height): Constructor gets as a parameter the length and height dimensions of the matrix; it sets the length and height accordingly and creates the matrix structure

private void fillMatrixWithValue(T defaultValue): Private method that fills the matrix with a default value given as a parameter; it makes use of an enhanced for loop and a method from the Arrays class that resides in the java.util package

public String toString(): toString() method from the Object class has been overridden in order for it to return the length and height of the generic grid

11.Class ToyFactory

Implementation of a Santa Claus Toy Factory from the Workshop; this is achieved by extending the Thread class; this class does a lot of work: it spawns the elves, it decides their moves and supervises the gift creation process that is done by the elves; it contains a boolean grid, it uses a ReentrantLock to ensure that the movement of elves is correct and has a concurrent queue that plays the role of the deposit of gifts where all presents created by elves are stored awaiting for elves to take and afterwards pack and send them to Santa Claus trough a pipe; I have chosen to let each factory know the number of reindeer that will take gifts from them, and, to simplify a bit the structure of the program, the number of reindeer is 10 for each toy factory

`private static final int QUEUE_MAX_SIZE`: Constant that stores the maximum size of the gift queue

`private int factoryIndex`: Index of the current factory

`private int factoryGridSize`: Size of the toy factory grid

`private int numberOfWorkingElves`: Member that counts the number of elves that are currently working in the factory

`private SimpleBooleanGrid factoryGrid`: An instance of the SimpleBooleanGrid class, the grid of the toy factory

`private List<ElfWorker> workingElvesList`: List of all elves that are working in the factory (they are added immediately after spawning)

`private ReentrantLock gridlock`: A reentrant lock that blocks the access to the grid when moving an elf in order to avoid synchronization problems

`private ConcurrentQueue<ChristmasGift> giftQueue`: Concurrent queue that holds all the presents that have been created by elves; this is the place from which reindeer take afterwards the presents and pack them `private int numberOfAssignedReindeer`: Number of reindeer assigned to communicate with the factory in order to take presents from the queue and to pack them

`public ToyFactory(int factoryIndex)`: First constructor of the Toy Factory has a parameter represented by the factory index that is thereby set; it also sets the factory grid size to a default value of 100 (minimum accepted), the number of working elves to 0, creates an empty factory grid, an empty working elves list, creates the lock that will be used later for synchronization and the concurrent queue of gifts; it also sets the number of reindeer to 10 for the current factory

`public ToyFactory(int factoryIndex, int factoryGridSize)`: Second constructor of the Toy Factory has two parameters represented by the factory index and the factory grid size; it sets the factory grid size, the number of working elves to 0, it creates an empty factory grid, an empty working elves list, the lock that will be used later for synchronization and the concurrent queue of gifts; it also sets the number of reindeer to 10 for the current factory

`public void run()`: method from Thread is overridden in order to achieve the needed behavior; the toy factory spawns elves until their number is equal to half of the grid matrix order; it has a random spawning time between 500 milliseconds and 1 second; first of all, it finds a valid position to spawn an elf; then it creates an elf and gives him an index that is unique in the factory, tells him that he will work in this facility and puts it in the position that was empty; after creating the elf, he will notify factory of its position, in this way ensuring us that the process completed in a correct way and therefore that he can be now started; then it adds it to the list of working elves, increases the numbers of workers and starts the elf thread so that he can do its job; then it performs a sleep before continuing with the next elf spawning

`public void moveElfOnGrid(ElfWorker elfToMove)`: Method that performs the move command of an elf on the factory grid; first of all, this must be a synchronized action so that no two elves will try to move at the same time on the same position and creating a conflict; therefore, if lock for the movement has been taken, we get the current position from the elf that must be moved; then we get a list of available moving positions for the current elf (list of Strings of form "DOWN", "UP", "RIGHT", "LEFT")'; if there is no available moving position, the elf will sleep for 10 to 50 milliseconds; if, however, the elf can move, it gets a random direction from the list of available ones and performs a move towards that certain point; the movement is done by a method from the class and implies setting a new position, notifying the factory about the new position, creating a gift, sending it towards the queue and then resting for ashort while

public String toString(): toString() method from the Object class has been overridden in order for it to return the number of the factory and the grid size of the underlying matrix

private List<java.lang.String> getListOfAvailableMovingPositions(Position2D currentPosition): Method that returns a list of the available moving positions for an elf; we firstly generate all cardinal points from the current elf position; then, for each of them, we test if it is on the grid surface and if it is occupied by another elf or not; if a position is valid in both of the above checks, a string with the direction is saved in the array list;

private void performAnElfMoveUp(ElfWorker elfToMove): Method that moves an elf to the north; it firstly saves the new position by calling the notifyFactoryOfMoveUp() method, then it creates a new gift and passes it to the factory (it is added in the queue); afterwards, the elf gets to rest for a random time in milliseconds and his number of produced toys is increased by one

private void performAnElfMoveDown(ElfWorker elfToMove): Method that moves an elf to the south; it firstly saves the new position by calling the notifyFactoryOfMoveDown() method, then it creates a new gift and passes it to the factory (it is added in the queue); afterwards, the elf gets to rest for a random time in milliseconds and his number of produced toys is increased by one

private void performAnElfMoveLeft(ElfWorker elfToMove): Method that moves an elf to the west; it firstly saves the new position by calling the notifyFactoryOfMoveLeft() method, then it creates a new gift and passes it to the factory (it is added in the queue); afterwards, the elf gets to rest for a random time in milliseconds and his number of produced toys is increased by one

`private void performAnElfMoveRight(ElfWorker elfToMove)`: Method that moves an elf to the east; it firstly saves the new position by calling the notifyFactoryOfMoveRight() method, then it creates a new gift and passes it to the factory (it is added in the queue); afterwards, the elf gets to rest for a random time in milliseconds and his number of produced toys is increased by one

`private void createNewGiftInFactory()`: Method that creates a new gift in the factory, sets its packing member to false and then pushes it into the concurrent queue

`private Position2D findValidPositionToSpawnElf()`: Method that searches the grid for a position in which there is no elf and returns it so that a worker can be put there to perform its job; we start in a random position (we generate a random coordinate X and a random coordinate Y) and search to the end of the matrix; if we find a good value, we mark it as we will put an elf in it and then we return it; if no valid posiution has been found, we start another search, from the start of the matrix towards the random position; we have the certitude that we will finally get an empty position because the number of elves is way smaller than the number of positions in a troy factory grid

12.Class RandomGenerator

Implementation of a random number generator class which has some easy to use methods for getting a random value, a random value that is less than a given one, or a random value between a minimum and a maximum This is a simple object structure that is needed in the micro world of the program in order to simulate a North Pole Workshop; this is a functional class, that contains just utility methods; all classes have the same name, but different signatures, so they are overloaded in order to provide an easy to use form

`public int returnRandomValue()`: Method that does not take any parameter and returns a random integer value

`public int returnRandomValue(int maximumValue)`: Method that takes one parameter and returns a random integer value that is smaller than or equal with the maximumValue parameter

`public int returnRandomValue(int minimumValue, int maximumValue)`: Method that takes two parameters and returns a random integer value that is smaller than or equal with the maximumValue parameter, but greater than or equal with the minimumValue parameter

13.Class MainLauncher

Main class of the program, place in which we do only a few things in order to get the whole structure of classes up and running

`static void main(java.lang.String[] args)`: Main entry, the main of the project; we create a new NorthPoleWorkshop instance and start it