Juice Bottler Lab – Nevin Fullerton

Project Description:

This project simulates multiple juice bottler plants producing orange juice using multiple threads in Java. The project has multiple plants, which store the oranges and coordinate workers, and multiple workers, who are specialized in processing a specific orange state. The goal is to produce as many oranges as possible, while maintaining thread safety and avoiding race conditions.

Project Requirements:

The project meets the requirements determined by the lab assignment document and grading criteria.

Multiple Plants

There are at 10 plants by default, which can be changed to another value, that all have separate oranges and workers. Their data are completely independent of each other.

Multiple Workers

Each plant has multiple workers, which can also be changed to any amount desired. Each worker is assigned to a specific production stage (fetch, peel, squeeze, etc.). They only work on oranges that are in the state they are assigned to, an example of task parallelization.

Thread Safety

This program uses multiple locks to prevent multiple threads from accessing shared resources, lists, and integers in this example, preventing race conditions.

Naming and Formatting

The program uses the proper java naming convention and is properly formatted for easier reading.

Summary

The program summarizes the results of the plants by adding up all of their stats (oranges provided, oranges wasted, etc.) and outputs them after the program is done.

Ant

The source is built using Apache Ant.

How to Run

Prerequisites

Java 21
Apache Ant

Running

1. Pull down the GitHub repository into the local repository.
2. Type ‘ant run’ into the console. This will compile and run the program.

Challenges faced:

There were multiple challenges during the course of this project.

First Challenge – Creating Worker Threads:

My first major challenge was figuring out how to handle the worker threads when they are created, and how they should work. My solution was to make a separate ‘worker’ class that would process the entire orange, from a shared array in the plant class. After creating the class, I then had to figure out how the thread can access the oranges that are inside the plant. This was solved by passing a reference of the plant to the thread, then creating two methods in the plant class named “RequestOrange” and “ReturnOrange”, allowing the thread to request an orange to work on, and return it when it's done processing the step. The oranges were originally stored in a fixed array, with all threads picking an orange that was available, locking that orange, processes the step, unlock the orange, than find another orange. This worked fine for multiple worker threads, but it did not implement task parallelization and was inefficient. So my next step was to specialize the workers in a specific task.

Second Challenge – Task Parallelization:

Implementing task parallelization seemed straightforward, but came with many small complications that required an almost full rework of the code. The first problem was how to specialize the workers. I accomplished this by using Orange.State class. In the constructor, I passed the state the worker would be working in. For example, if I passed the state Orange.State.Peeled, that worker would only work on oranges that were peeled. I changed the data structure from an array to multiple lists, each list for each state of the orange (except for fetched and processed). Once a worker did work on the oranges, they would be returned to the plant and put into the next list, waiting for another worker to process them. Though this led to another problem, because I designed the methods to only have one lock for the entire method. This worked fine for one array, but when workers only need to access one list of multiples, I needed a way for multiple workers to request and return oranges if they were not interfering with each other. So this lead to the next challenge.

Third Challenge – Multiple Locks:

When redesigning the “RequestOrange” and “ReturnOrange” method, I had to figure out how to allow multiple threads from different production lines to not interrupt each other. I solved this by

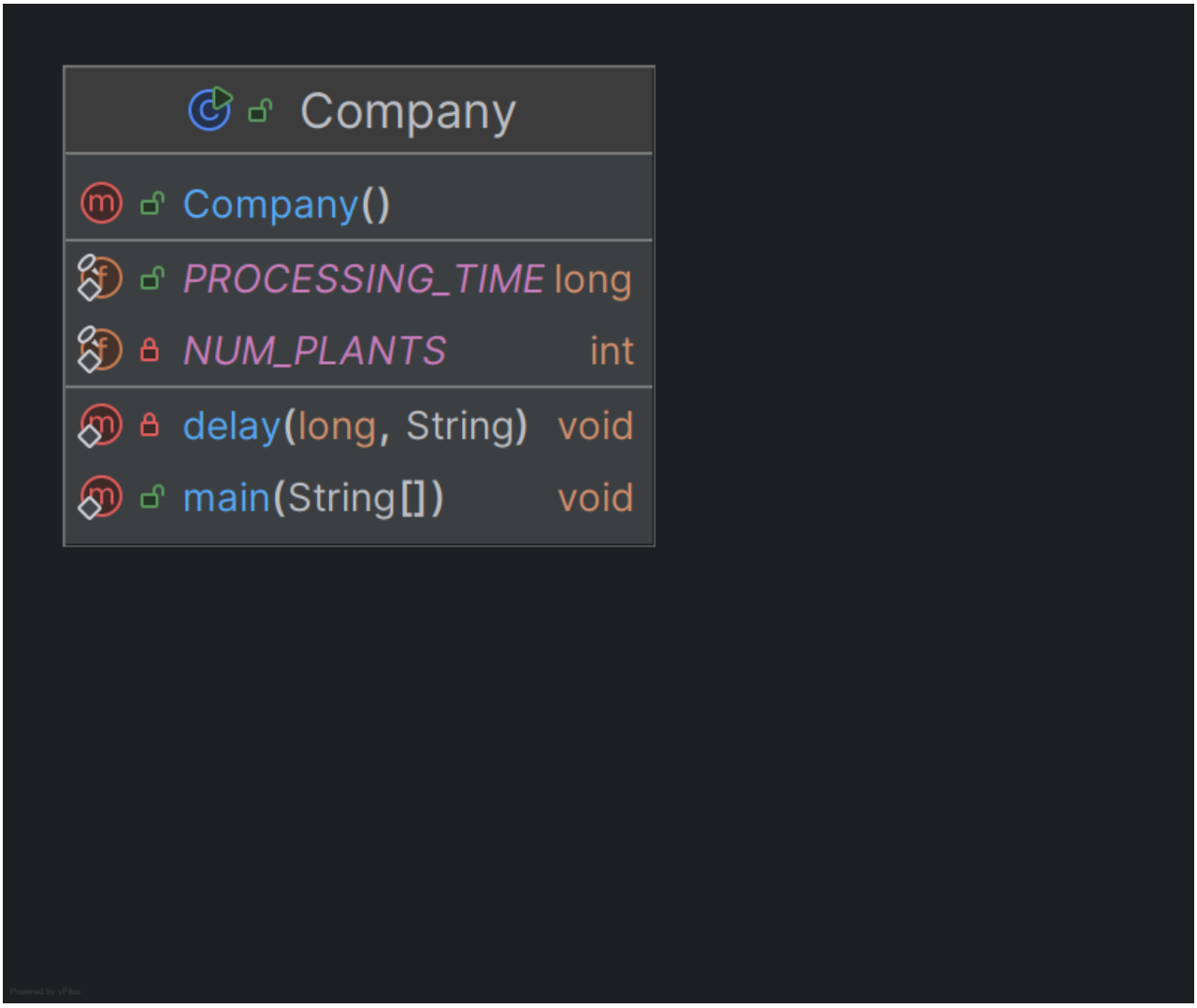
creating a lock for each orange state, and in the methods, have each list as a critical section so that only workers of that line will have to wait for other workers of the same line, rather than one worker blocking all workers. This also solved the issue of returning and requesting to the same list at the same time, since the list in both methods have the same lock on them, preventing race conditions. But there was one final issue, and that was a deadlock that stopped the program.

Fourth Challenge – Deadlock:

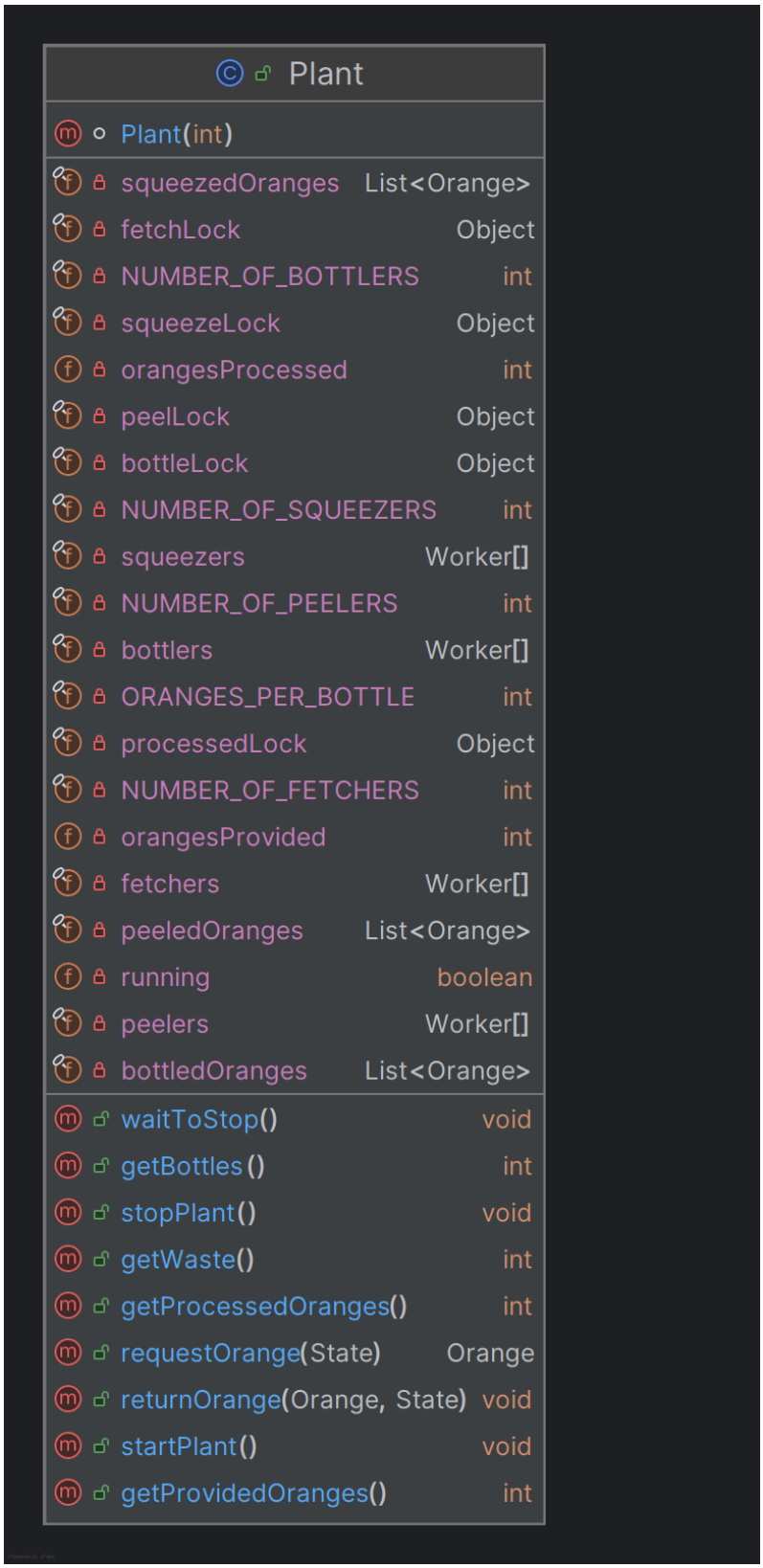
When I finished fixing the exceptions, I ran into the final major obstacle, a deadlock. Assuming the deadlock came from “notifyall” and “wait” not working properly, I investigated a problem with the logic when starting the plant. Not finding a logic problem, I investigated other parts of the code, and realized the problem was actually when stopping the workers. The workers were operating fine when the plant was running, but when the order to stop was given, the workers who were not already working on an orange would get stuck waiting in the request orange loop, waiting for an orange that will not appear. This deadlock happened because there was no secondary way for workers waiting to be signaled, except when a worker put an orange in the list, causing them to wait for ever. Even if they were notified, they would still check for orange and go straight back to waiting. To fix this, I implemented a new boolean in the “requestOrange” while loop that would check if the plant was stilling running, if it was not, then don’t look for an orange and leave the loop without an orange. Then, when I stopped the workers, I also notified all workers in every lock to ensure no worker would be waiting as the plant shut down. Finally, the plant worked fine and could produce orange juice effectively and safely.

UML

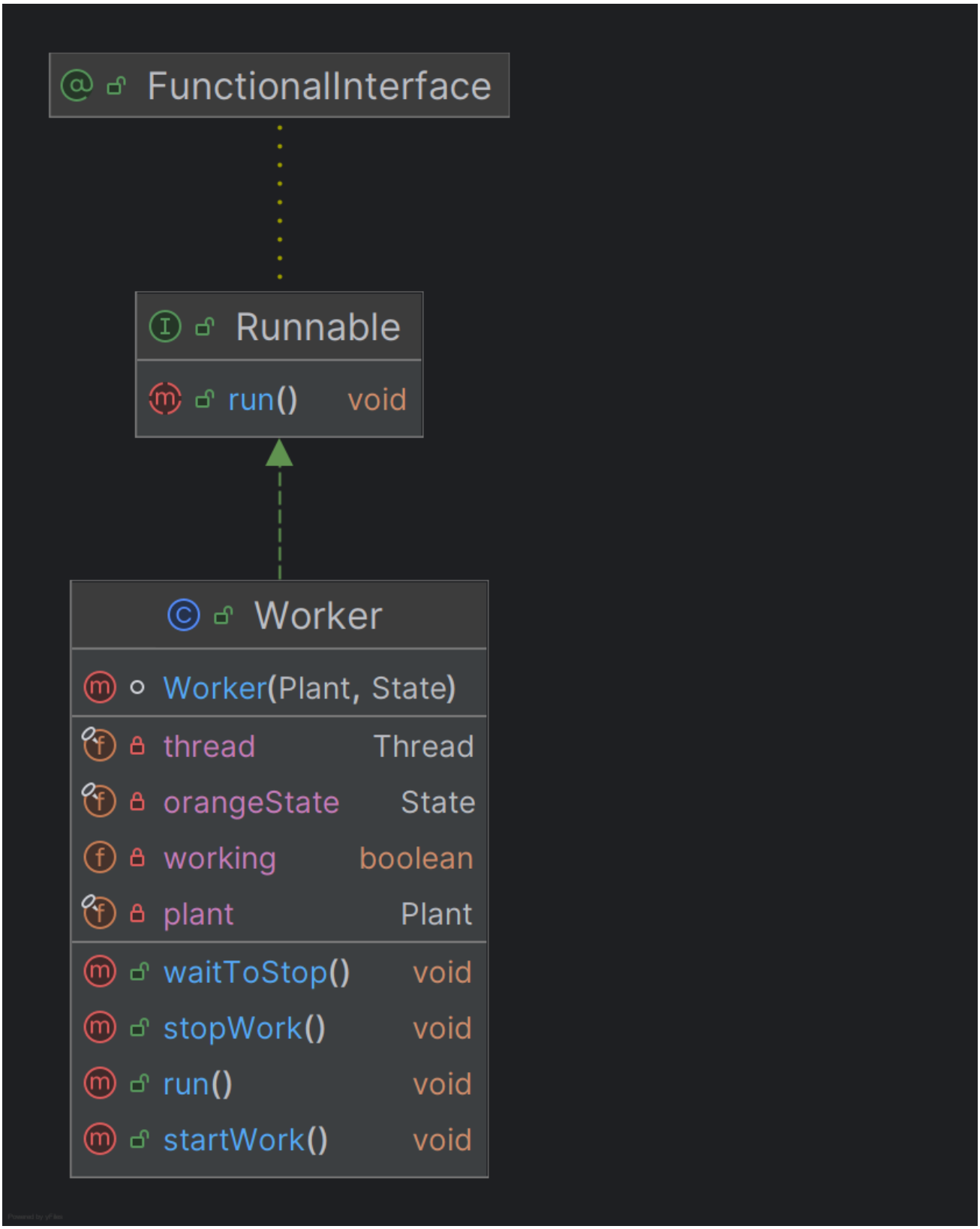
Company


Company
Company()
PROCESSING_TIME long
NUM_PLANTS int
delay(long, String) void
main(String[]) void

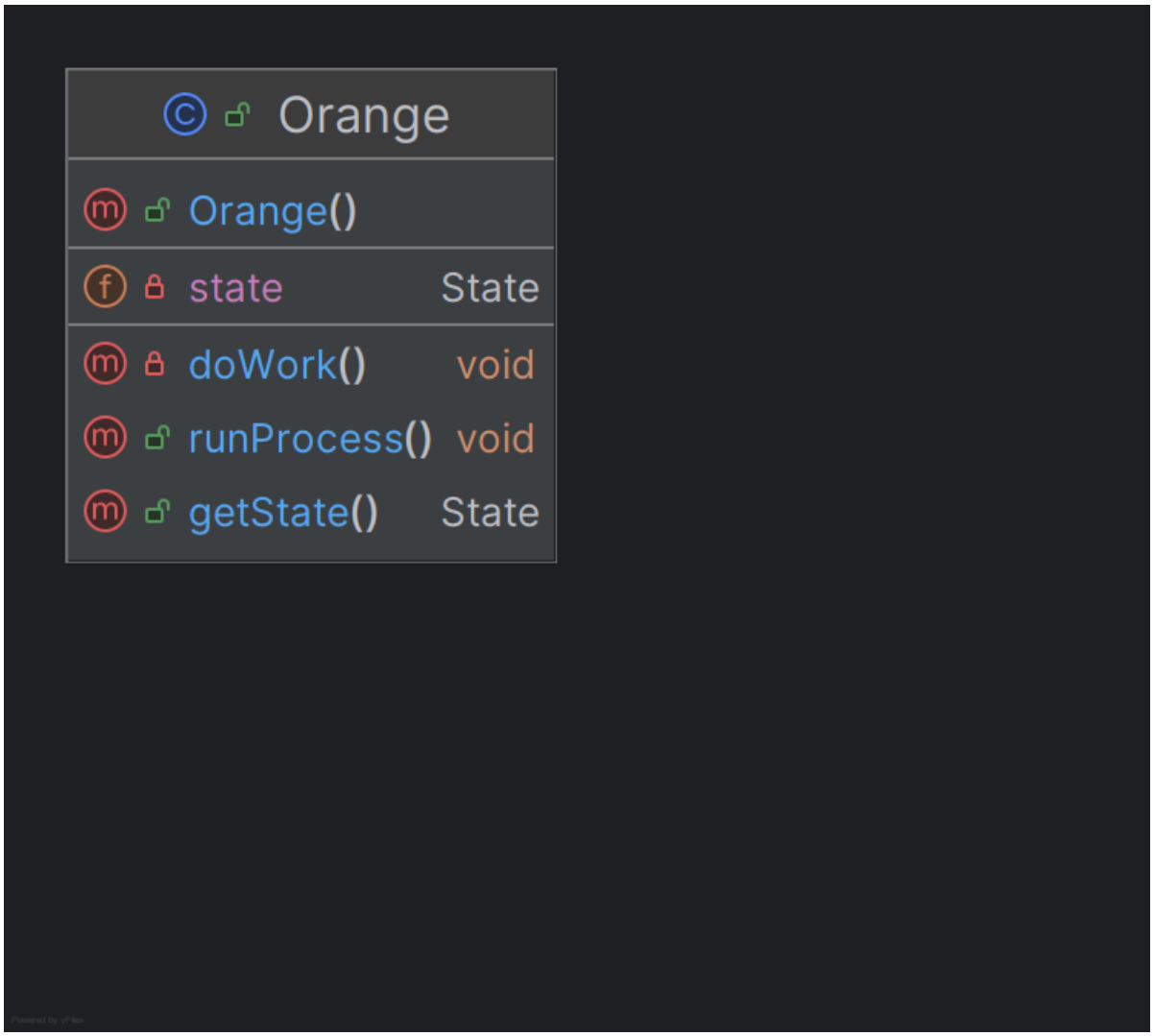
Plant


Plant
Plant(int)
squeezedOranges List<Orange>
fetchLock Object
NUMBER_OF_BOTTLERS int
squeezeLock Object
orangesProcessed int
peelLock Object
bottleLock Object
NUMBER_OF_SQUEEZERS int
squeezers Worker[]
NUMBER_OF_PEELERS int
bottlers Worker[]
ORANGES_PER_BOTTLE int
processedLock Object
NUMBER_OF_FETCHERS int
orangesProvided int
fetchers Worker[]
peeledOranges List<Orange>
running boolean
peelers Worker[]
bottledOranges List<Orange>
waitToStop() void
getBottles() int
stopPlant() void
getWaste() int
getProcessedOranges() int
requestOrange(State) Orange
returnOrange(Orange, State) void
startPlant() void
getProvidedOranges() int

Worker


FunctionalInterface
Runnable
run() void
Worker
Worker(Plant, State)
thread Thread
orangeState State
working boolean
plant Plant
waitToStop() void
stopWork() void
run() void
startWork() void

Orange


Orange
Orange()
state State
doWork() void
runProcess() void
getState() State