Main

Concurrent programming Lab exercises

Java recap

1. Simple Java class

   Create a Java class called `Employee`. An `Employee` has two private fields, called `name` and `salary`. Create a public constructor that initializes these fields. Also create public getters for these fields and a method that raises the salary by a given percent (a floating point value passed as parameter).

   In this exercise and all the other ones, create a class whose `main` method tests the described functionality. Make sure that you can compile the code and run it.

2. Abstract class and inheritance

   Change the `Employee` class to an abstract class. Also create two subclasses called `Manager` and `Subordinate`. Make the getter for the salary field in the `Employee` class abstract and copy its original logic into an override in the `Subordinate` class. For the `Manager` class also store a list of `Employee`s. Create functions for adding and removing `Employee`s of a `Manager`. Override the `getSalary()` function of a `Manager` to return the sum of the own salary of the manager plus 5% of each of its `Employee`s' salary.

3. Interface

   Create an interface called `SalariedEntity` with a single `getSalary()` function. Change the class `Employee` to inherit from this interface. Also create a class `Subcontractor` which also implements this interface. Instead of a name, `Subcontractor`s have a tax number (`long`).

4. 🏠 Homework

   Create a new class called `Company` which contains a list of `SalariedEntities`. Create functions to add a new entity, delete entities and also one which raises all the salaries by a specific percent, but only for the `Employees`.

Creating threads

1. Make two threads and run them. The threads will print the texts `hello` and `world` lots of times (e.g. `10_000`) using `System.out.println()`.

   Create a child class of `Thread` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Thread.html) , and write your implementation into its `run()`. To start the threads, make two instances of your new class and call `start()` on them.

   a. Notice that the outputs are interleaved.

      a. Make another solution that prints letter by letter, using the `print()` method of `System.out`, not `println()`.

   b. Try what happens if you run the above by invoking the `run()` method instead of `start()`. This way, no new execution threads will be started.

   c. Also create a solution where you write the output to a file instead of the standard output using `PrintWriter` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/io/PrintWriter.html) .

   d. Make other solutions by creating `Thread` objects in different ways. In each case, consider whether it is possible to write the code in such a way that the common part of the action of the two threads (to print something) is not repeated in the code, but appears only once, and the threads parameterize this by specifying exactly what text to write out.

      - using a child class of `Thread`
      - using a class that implements the `Runnable` interface
      - using an anonymous class derived from `Thread`
      - using an anonymous class derived from `Runnable`
      - by passing a lambda to the `Thread` constructor

2. Experiment with threads.

   - Name the threads using `setName` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Thread.html#setName(java.lang.String)) .
   - Create a `ThreadGroup` (https://download.java.net/java/early_access/loom/docs/api/java.base/java/lang/ThreadGroup.html) instance and pass it as the first argument at thread instantiation.
     - Using the `activeCount()` (https://download.java.net/java/early_access/loom/docs/api/java.base/java/lang/ThreadGroup.html#activeCount()) and `list()` (https://download.java.net/java/early_access/loom/docs/api/java.base/java/lang/ThreadGroup.html#list()) methods, observe the execution of the threads.
   - ➿ Optional: use an IDE to suspend threads in *debug* mode.
     - Examine the contents of their variables using the debugger.
     - Rewrite the values of the variables and continue running.

3. 🏠 Homework: check that execution using multiple threads causes significant speedup.
    - Add up the numbers of the interval `1..1_000_000_000` on a thread. Measure how long it takes using `System.nanoTime()` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/System.html#nanoTime()) .
    - Then start `10` threads, which add up the numbers of the interval `1..1_000_000_000`:
        - The first thread works on the interval `1..100_000_000`, the second thread works on the interval `100_000_001..200_000_000` etc.
        - Put the sum into a `static` variable.
        - Calculate how long it takes using `System.nanoTime()` and print it at the end of the execution of the thread.

4. 📺 Demo: Thread operations: start, join, sleep, interrupt

    Running `DemoBasicThreading.java` showcases all examples. Each functionality has its own file, and `DemoBasicThreadingHelper.java` contains helper functions used by multiple examples.

    1. Class extending Thread: `ClassExtendingThread.java`
    2. Class implementing Runnable: `ClassImplementingRunnable.java`
    3. Thread Creation: `ThreadCreationExample.java`
    4. Join, Sleep & Synchronized: `JoinSleepSynchronizedExample.java`
    5. Interrupting a Thread: `LifecycleInterruptExample.java`

5. Join, sleep, interrupt

    a. Change the **Make two threads…** program to print the message `ready` after both threads finish.
        - For a quicker finish, you may lower the constant to `1000`.
        - Use the `join` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Thread.html#join()) method of class `Thread` to wait for a thread to finish.
            - Do not forget to catch `InterruptedException` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/InterruptedException.html) .

    b. Waiting inside Threads
    Let the program to wait 5 milliseconds between printouts using the static method `sleep` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Thread.html#sleep(long)) of class `Thread` to wait inside a thread for a specific amount of time.
        - You will have to catch `InterruptedException` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/InterruptedException.html) .
        - Note that there is also a method called `wait()` which is completely different.

    c. Interrupting Threads
    Interrupt both of the threads after a second. Use the `interrupt` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Thread.html#interrupt()) method of class `Thread`.

d. Restarting Interrupted Threads
Try to restart your threads after you interrupted them. What happens?

6. Join, sleep, interrupt #2

- Start 10 threads, all counting from `1` to `1_000_000` and print each number to a file (`PrintWriter` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/io/PrintWriter.html)) with the same name as the thread name (`getName()` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Thread.html#getName())).
- Let the method that starts the threads wait 1 second (`Thread.sleep` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Thread.html#sleep(long))), then write the last line of the files associated with the threads to the standard output. Do you notice anything strange?
- Modify the solution to *join* the threads before final printout. To do this, call `join()` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Thread.html#join()) on each thread.
- The `Thread.sleep` and `Thread.join` operations may throw checked exceptions, which must be handled in an appropriate `try-catch` block. For now, it is sufficient to create an empty `catch` branch, no “exception handling logic” is necessary.

7. 🏠 Homework

Extend the solution to “**Joining threads**” by adding a new thread that prints out how many threads are still active every second (`Thread.currentThread().getThreadGroup().activeCount()`). If this is the only remaining thread, you may stop checking and let the program end.

- Use `Thread.sleep` for waiting.
- ➿ Optional: create another solution using the `scheduleAtFixedRate` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Timer.html#scheduleAtFixedRate(java.util.TimerTask,long,long)) method of the `Timer` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Timer.html) class.

8. 🏠 Homework: Miner & Builder Simulation

The goal of this simulation is to have two types of workers (represented by threads) who are trying to build 5 houses – the miners are mining gold from the goldmine, while the builders, once they have enough gold to do so, build houses.

- `Configuration.java`: This class contains the configurations for the simulation, such as duration of building a house, number of workers, etc. During completion this class should not be modified.
- `Resources.java`: This class keeps track of all the resources used by the simulation – goldmine capacity, gold owned by the workers and number of houses built. During completion this class should not be modified.
- `ThreadCraft.java`: This is the entry point of the simulation, containing the main method and behaviour of the workers. Implement the following:

    - Start the workers' threads (miners and builders) with the provided action (`mineAction()` and `buildAction()`)
    - Start a thread responsible for periodically logging the state of the simulation (`loggingAction()`)
    - In the main function, make sure that the workers' threads finish up before finally printing "Simulation over"
    - Implement the `sleepForMsec(int msec)` method which is used to invoke sleep on a thread for a given time

  Upload only the `ThreadCraft.java` source file.

`synchronized`

1. Shared data

   Start two threads that access the same list. Make one put the odd numbers from `1..1_000_000` into it, and make the other put in the evens. Don't synchronize access yet.

   - Wait for both operations to complete, then observe the following surprising phenomena:
     - How many elements does the list have? (It should have exactly one million.)
     - Are the elements in order?
       - Hint: use `subList` ↗ (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/List.html#subList(int,int)) to explore a section of the list.
       - Or: print the number of inversions (a smaller value follows a larger value) in the list.
     - If you run the program a couple of times, you may even see an exception.
   - Now use a `synchronized` block around the call where you add an element to the list with the list itself as the lock.
     - How does the content of the list change?
     - How about its element count?
     - How about the order of the elements?
   - ➿ Optional: modify the insertion so that the elements end up in order. To achieve this, widen the scope of the `synchronized` block by including an `if` condition that makes sure that only the very next element can be inserted. Pay attention to not leaving out any value.
     - How does this change affect the execution time?

2. 🏠 Homework

   Generalise the "**Shared data**" exercise in the following ways by adding some parameters.

   - Instead of using two threads, use `n` of them. Let each put every `n`th value of the range into the list.
   - Let the boolean value `isSynchronized` dictate whether to use synchronization around the insertion call or not.
   - Let the boolean value `isInOrder` dictate whether the values should be inserted in a strictly increasing order.

   - ꝏ Optional: write the code so that functionality is not duplicated.
     - Hint: put the general code in a helper method, and call it with the appropriate parameters.
     - Hint: the helper method returns a `Runnable`.
   - Make a method `useThreads` that takes `n`, `isSynchronized`, and `isInOrder` as arguments. This method contains the code that was previously in `main`.
   - In the actual `main`, call `useThreads` a couple of times with various sets of parameters. Use `System.nanoTime()` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/System.html#nanoTime()) to measure and print execution times.

3. Create class `ThreadSafeMutableInteger`, a simplified version of `java.util.concurrent.atomic.AtomicInteger` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/atomic/AtomicInteger.html).

   a. Implement the `ThreadSafeMutableInteger` class which contains an `int` field (you’re not allowed to use `AtomicInteger` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/atomic/AtomicInteger.html) itself), and provides thread-safe read/write operations on it. Use the `synchronized` keyword (either on the methods or as a block). Let the following be part of the class:

      - two constructors: `ThreadSafeMutableInteger()` and `ThreadSafeMutableInteger(int)`
      - basic read/write operations: `int get()` and `void set(int)`

   b. In `main`, create a shared `ThreadSafeMutableInteger` and 10 threads that each increment its value `10_000_000` times. When all is done, print the result.

   c. Implement the following atomic operations.

      - `int getAndIncrement()`
      - `int getAndDecrement()`
      - `int getAndAdd(int v)`
      - `int incrementAndGet()`
      - `int decrementAndGet()`
      - `addAndGet(int v)`

   d. Also write the following code to ensure the correct operation of the other methods. Make a separate `main` for each use case, and check that the end result is `0`.

      - Let half of the threads use `getAndIncrement`, the other half use `getAndDecrement`.
      - Let half of the threads use `addAndGet` by `+2`, the other half by `-2`.

   e. In the previous exercise, make a version that uses a `get` and a `set` operation in sequence in place of the `getAndIncrement`/`getAndDecrement` calls. What is the outcome? Why is it so?

   f. ꝏ Optional: implement the following methods that allow user defined, atomic value transformations. Validate your solution in a new `main`.

        - `int getAndUpdate(IntUnaryOperator)`
        - `int updateAndGet(IntUnaryOperator)`

4. 🏠 Homework: create class `ThreadSafeMutableIntArray`, a simplified version of `java.util.concurrent.atomic.AtomicIntegerArray` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/atomic/AtomicIntegerArray.html) .

    a. Create the `ThreadSafeMutableIntArray` data structure that implements a thread-safe array of `int`s.

    - All elements of the `ThreadSafeMutableIntArray` are initialised to `0`.
    - As a representation, use `int[]`. You're not allowed to use `AtomicIntegerArray` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/atomic/AtomicIntegerArray.html) itself.
    - Use an `int[]` field in the representation.
    - ⚠️ Important: make sure to let different threads access separate elements without blocking.
        - You are therefore **not allowed to use** `synchronized` **methods** as that would lock the whole `ThreadSafeMutableIntArray`.
        - Rather, create an array of `Object`s called `locks`. Using these, the elements will be locked individually.

    b. Define the following.

    - Constructor: `ThreadSafeMutableIntArray(int capacity)` which creates the backing array of size `capacity` and `locks` of similar size.
        - While the `int[]` is initialised right by default, you will have to create and insert `Object` instances into `locks`.
    - `int get(int idx)` and `void set(int idx, int newValue)`: these operations use synchronization with the `locks` object at index `idx`.

    c. Validate your solution in `main`: create 10 threads and a shared `ThreadSafeMutableIntArray` with two elements. Let half of the threads write the first element, and half the second one.

    - Let each thread increase its target up until `10_000_000`.
    - Print the results to the standard output.
    - Do you experience anything strange? Can you do something about it?

    d. ➿ Optional: implement the following methods that allow user defined, atomic value transformations. Validate your solution in a new `main`.

    - `int updateAndGet(int n, IntUnaryOperator)`
    - `int getAndUpdate(int n, IntUnaryOperator)`

`ExecutorService`

1. Let us represent a system where some clients take loans from a bank.

Create a thread pool using Executors.newFixedThreadPool (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/Executors.html#newFixedThreadPool(int)) . Use submit() (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/ExecutorService.html#submit(java.lang.Runnable)) to add threads representing the clients into this pool.

The clients take loans in many rounds (e.g. 10000), each time using ThreadLocalRandom.current().nextInt(min, max) (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/ThreadLocalRandom.html#nextInt(int,int)) to determine the amount of the loan (which should be between 100 and 1000). The bank has a (properly synchronized) counter in a variable that always shows the total amount of loans taken. The clients themselves store how much loan they have taken, and at the end of their execution, they write this number into the appropriate element of an array.

At the end of `main`, use shutdown() (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/ExecutorService.html#shutdown()) and awaitTermination() (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/ExecutorService.html#awaitTermination(long,java.util.concurrent.TimeUnit)) to wait for all client threads to finish. Once they're all done, print the bank's counter and also print the sum of the loans in the client array. The two numbers should match.

2. We solve the same problem as before but now we use a different submit() method (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/ExecutorService.html#submit(java.util.concurrent.Callable)) .

   At the end of the client thread's code, do not write the loan amount into an array. Instead, let it be the return value of the anonymous function that is passed to `submit()`. The return of `submit()` is a Future (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/Future.html) , which has a get() (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/Future.html#get()) operation that gives you the computed loan amount.

   This time, invoke `shutdown()` on the pool only after summing/comparing the loans in the two different ways.

3. 🏠 Homework: let `main` in class `Switcheroo` start 10 threads. All threads have shared access to an array of 100 elements, initially all set to 1000. The threads repeat the following 10000 times: choosing two indexes at random, a random amount is transferred from the first index to the second one (decreasing the value at the first index to not lower than zero). Make sure that the transfer is properly synchronized.

   After all threads are done, check whether the total sum of the array remains.

4. 🏠 Homework: let `main` in class `InefficientSorter` start 10 threads. All threads have shared access to an array of 100 elements, initially all set to random values. The threads repeat the following 10000

times: choosing two indexes at random, they are swapped if the lower index contains the higher number. Make sure that the transfer is properly synchronized.

With a high probability, this will result in a sorted array. Check whether it is so, and further check whether the resulting array contains exactly the same elements (in a different order) as it did in the first place.

Synchronized Data Structures

1. Synchronized Lists

   a. Create a method called `nonSyncIterate` which iterates on the elements of a a `java.util.Collection<Integer>` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Collection.html) using `Iterator<Integer>` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Iterator.html) and prints the elements. The method should have an additional parameter, a number which should also appear in the printing. Also create a method called `syncIterate` which calls the former one in a `synchronized` block. The lock object must be the collection.

   Create the list the following ways, using a lot of (e.g. 100_000) elements:

   - Simple `ArrayList`
   - Simple `LinkedList`
   - Simple `Vector`
   - A synchronized data structure `Collections.synchronizedCollection` created from one of the above
   - A synchronized data structure `Collections.synchronizedList` created from one of the above

   Try the following with as many combinations as possible of the various methods and data structures and inspect the structure of the output.

   Start two threads. Both threads must call the chosen method and pass the reference to the list and its own number.

   b. 🏠 Homework: we have two shared lists, `original` and `result`. The second one is empty at the start, the first one is filled with `1, 2, ...`, the last value being `THREAD_COUNT * ELEMS_PER_THREAD` where `THREAD_COUNT` is at least 2 and `ELEMS_PER_THREAD` is quite large (say, `100_000`).

   - Let an `ExecutorService` run `THREAD_COUNT` tasks that repeat the following `ELEMS_PER_THREAD` times: `remove` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/List.html#remove(int)) the first element of `original` and `add` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/List.html#add(int,E)) it to the beginning of `result`.
   - Stop the `ExecutorService` and print the first 100 elements of `result`.
     - Most executions will end with a `ConcurrentModificationException`.

- Try the same with synchronized lists. The exception should go away.

2. `ConcurrentMap`

   a. We have a calendar where we schedule meetings. One meeting takes 10 minutes and must not conflict with each other. 10 threads schedule 5000 meetings per thread into the calendar while another 10 threads delete 2500 meetings per thread from the calendar. There is a 21st thread which looks for and prints the next meeting every 10 milliseconds.

      Since our program utilizes the plain `java.util.HashMap` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/HashMap.html) to store the meetings, which is not thread-safe, it does not terminate in most cases but throws a `ConcurrentModificationException` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/ConcurrentModificationException.html) . Fix the program. To achieve this, use method `java.util.Collections.synchronizedMap()` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Collections.html#synchronizedMap(java.util.Map)) which we already used earlier and also do not forget to manually synchronize the iteration.

   b. Modify the previous solution where you replace method call to `java.util.Collections.synchronizedMap()` with a new data structure `java.util.concurrent.ConcurrentHashMap` (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ConcurrentHashMap.html) . Also remove the synchronized block surrounding the iteration because using this type it is not necessary anymore. Observe (by counting the printed lines) the speed difference between the two solutions.

   c. Create a program which uses some synchronized version of `Map`: call insertions, deletions and iterations from many threads. You program should be as simple as possible, you do not have to make it as complicated as the solution of the previous assignment.

      Demonstrate with your program that for some values of the parameters (e.g. the number of threads or the elements to be inserted into the `Map`) the `synchronizedMap()` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Collections.html#synchronizedMap(java.util.Map)) but for other values of the parameters `ConcurrentHashMap` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/ConcurrentHashMap.html) is the faster.

   d. 🏠 Homework: simulate a stock exchange. The prices (a floating point number) of stocks (3 capital letters) are stored in a map, each stock starts from $100. We have 100 brokers who randomly buy and sell stocks for 10000 rounds. If a broker buys a stock, its price goes up by 1%, if he/she sells it, then it goes down by 1%. There is also a separate thread that periodically (1 secs) prints the actual stock prices. Use `Collections.synchronizedMap()` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Collections.html#synchronizedMap(java.util.Map)) to ensure thread safety of the map.

e. 🏠 Homework: change your previous solution to use [ConcurrentHashMap (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Collections.html#synchronizedMap(java.util.Map))](https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Collections.html#synchronizedMap(java.util.Map)) . Since stocks are independent of each other, do not unneccessarily synchronize the iteration.

3. `BlockingQueue`

You can download the skeleton for the following exercises under the name `PipelineN.java` (`N`=1,2,3).

a. Create two `BlockingQueue` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/BlockingQueue.html) s: the first carries texts, the second one carries numbers.
   - Use `ArrayBlockingQueue` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/ArrayBlockingQueue.html) or `LinkedBlockingQueue` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/LinkedBlockingQueue.html) for instantiation.
   - Create three threads. The firs one submits some texts to the first queue.
   - Alternatively, you may read the texts from a file.
   - Another thread reads the texts from the first queue and puts their lengths into the second one.
   - A third thread reads the numbers from the second queue and writes them to the standard output.
   - Put a special terminator element (`""`, `Integer.MAX_VALUE`) into the queues as the last element to indicate that no more elements are forthcoming.
   - To better distinguish these terminators, they should have clearly marked variable names such as NO_FURTHER_INPUT1 and NO_FURTHER_INPUT2.
   - Finally, stop the thread pool that handles the threads of the exercise.

b. Make a pipeline with many components.
   - We have many functions (all of them use different formulas) that look like this: `Function<Integer, Integer> fun = n -> 2 * n + 1;`
   - The type `java.util.function.Function` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/function/Function.html) has to be imported to use it.
   - Invoke the function with an argument like this: `fun.apply(123)`
   - The pipeline has a first stage, a last stage, and several intermediate stages.
   - The first stage takes some numbers and puts them into the first `BlockingQueue` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/BlockingQueue.html) .
   - The intermediate stages take the incoming numbers from the appropriate `BlockingQueue` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/BlockingQueue.html) , invoke the appropriate function, and put the result of the computation into the next `BlockingQueue`

(https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/BlockingQueue.html) .

- The final stage prints the elements coming out of the last queue.

c. Create a pipeline to filter primes.

- The pipeline has stageCount components. `BlockingQueue` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/BlockingQueue.html) s connect neighbouring stages, and there is also a final `BlockingQueue` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/BlockingQueue.html) .
- The first `BlockingQueue` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/BlockingQueue.html) gets the numbers 3, 5, 7, … up to a given upper limit, and finally Integer.MAX_VALUE to indicate the end of the input.
- To represent the stages, put a total of stageCount Callables into a thread pool that do the following.
- Each Callable uses two neighbouring `BlockingQueue` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/BlockingQueue.html) s.
- The stage takes the first incoming number. This will be the prime of the stage.
- The stage then takes all other incoming numbers.
- If it finds Integer.MAX_VALUE, it sends it on, and then the stage is done.
- Otherwise: if prime divides the number, it gets filtered out (it is put into a local list). If there is a remainder, the number is possibly a prime, so it is placed into the next `BlockingQueue` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/BlockingQueue.html) .
- At the end of the stage, the Callable returns the list of filtered numbers.
- A thread pool starts all stages using invokeAll. Then get their results (the filtered out numbers) and print them.
- Also print the remaining elements, which are (almost) guaranteed to be primes.
- The output should look like this.

```
[9, 15, 21, 27, 33, 39, 45, 51, 57, 63, 69, 75, 81, 87, 93, 99]
[25, 35, 55, 65, 85, 95]
[49, 77, 91]
[]
[]
[]
[]
Remaining: [23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

- If the Callables also store the values of their primes in an array, the printout can look even better.

```
Filtered by 3: [9, 15, 21, 27, 33, 39, 45, 51, 57, 63, 69, 75, 81, 87, 93, 99]
Filtered by 5: [25, 35, 55, 65, 85, 95]
Filtered by 7: [49, 77, 91]
Filtered by 11: []
Filtered by 13: []
Filtered by 17: []
Filtered by 19: []
Remaining: [23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

1. The Sleeping Barber Problem

    a. A barber has one barber's chair in a cutting room and a waiting room containing a number of chairs in it. When the barber finishes cutting a customer's hair, he dismisses the customer and goes to the waiting room to see if there are others waiting. If there are, he brings one of them back to the chair and cuts their hair. If there are none, he returns to the chair and sleeps in it. Each customer, when they arrive, looks to see what the barber is doing. If the barber is sleeping, the customer wakes him up and sits in the cutting room chair. If the barber is cutting hair, the customer stays in the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits their turn. If there is no free chair, the customer leaves.

    The simulation of the barber shop is partially implemented in `BarberShop.java`. Your task is to fill in the missing parts which are the barbers sleep and the customers wake up tasks. Implement them using the methods `wait()` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Object.html#wait()) and `notify()` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Object.html#notify()) .

    b. Enhancement

    Interrupting the thread `Barber` is not elegant. Instead, use method `wait(long timeoutMillis)` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Object.html#wait(long)) to limit the length of the barber's sleep.

2. 🏠 Homework: Custom Blocking Queue

    a. Implement a very simple blocking queue using the wait & notify technique. Use one our earlier assignments to test it. It is enough to implement the `take()` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/BlockingQueue.html#take()) method. Since the queue is not aware of the threads waiting for the queue, use the `notifyAll()` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Object.html#notifyAll()) method to wake up any thread. Extend interface `Queue` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Queue.html) and any of its implementations, e.g. `ArrayDeque` (https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/ArrayDeque.html) .

Exercise

1. In this exercise, we create the representation of an auction house that employs modern artists who sell their works using **Non-fungible tokens** (https://en.wikipedia.org/wiki/Non-fungible_token) ,

abbreviated as NFT. We will handle these tokens, the works do not appear directly.

In the text, uppercase names are constants that are found in the downloadable `AuctionHouse.java`. In the same file, you will find the `runChecks()` method that main calls as the last step. This is the tester utility. At the end of the file, there are two helper methods which you can use: `sleepForMsec` lets the program do nothing for the given amount of milliseconds, and `getRandomBetween` can generate random numbers in an interval.

a. `Artist`

   In this task and all the others, write code that avoids bad things like race conditions and deadlocks.

   You will have to fill in the TODOs in `AuctionHouse.java` based on the description.

   Main will create and run the artists (and all the other participants of the system, to be described in the following exercises) on a thread each, and then waits for everyone to finish up. If you write your program well, all actors will stop working by themselves.

   We employ `ARTIST_COUNT` artists. They create works of art every 20 milliseconds with a price randomly chosen between 100..1000, and they put it in the first available element of the `nfts` array. If there is no more space left there, or if the total cost of the NFTs would exceed `remainingNftPrice`, then the artist stops working.

b. 🏠 Homework: `Auctioneer`

   The auctioneer works as long as there is at least one active artist, and once they are all done, he does 100 more auctions.

   - Use the `isAlive()` method (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.html#isAlive()) to detect whether the other threads are alive.

   All auctions begin with the auctioneer choosing one of the already created NFTs, and it creates a `BlockingQueue` in the variable `auctionQueue`. In a later task, art collectors will send their `AuctionBid`s in it.

   - In this Task, nobody will send bids, but we already prepare ourselves for the eventuality.
   - The auctioneer checks if a bid is incoming on `auctionQueue`. He does it at most `MAX_AUCTION_BIDS` times, and he is willing to wait at most one millisecond each time. If the bid has a higher sum than all previous ones, then this is considered to be the best bid so far.
     - Note that especially at the beginning, it can happen that no NFTs are available yet. Obviously, it’s impossible to begin an auction like that.
   - If the waiting times out, or there have already been `MAX_AUCTION_BIDS` rounds, the auctioneer finishes the auction.
     - The auctioneer stores the name of the art collector with the winning bid. You only need to remember the names of those art collectors who have already won (at least once).

        - Increase `soldItemCount` by one.
        - Increase `totalCommission` by 10% of the total price: the base cost of the NFT plus the best bid.
    - Let’s have 3 milliseconds of “pause” between auctions: during this time, let `auctionQueue` be `null`.

c. 🏠 Homework: Art collector

Let there be `COLLECTOR_COUNT` art collectors with the names `Collector1`, `Collector2` etc. They continue bidding as long as the auctioneer’s working (that is, his thread is active).

The art collector sleeps randomly for `COLLECTOR_MIN_SLEEP`..`COLLECTOR_MAX_SLEEP` seconds before attempting to bid.

He then checks whether there is an ongoing auction (does `auctionQueue` hold a valid object). If not, he increases `noAuctionAvailableCount` and starts waiting again.

If there is an ongoing auction, he makes a bid with a price randomly chosen between `1`..`MAX_COLLECTOR_BID` except if he has already participated in this auction.

Assignments

1. Will be available later.

2. Will be available later.