Lab – Threads

In this exercise you are about to use the entire multitasking system capabilities and also experience the Executor API

Phase 1:

- In this phase you will define an *ArrayList* based stack which is
    1. Naturally not thread-safe
    2. Have unlimited amount of elements but cannot manage a negative index
- Then you will create a producer that puts values in the stack & a consumer that polls values
- Your main class will create 2 producers and 2 consumers and make them work a single stack instance simultaneously
- In order to complete this phase follow these steps:
    1. Move to *labs\lab 3\Phase1\exercise*
    2. Edit the *Stack* class
        - *pop()* method pops a value from the stack. Therefore should be thread-safe. It also must block a consumer if the stack is empty, print out the taken value and the consumer thread name
        - *push()* method takes an int and adds it to the stack. Therefore should be thread safe. It also releases the stacked consumer threads and prints out the inserted value and the producer thread name who puts it
    3. Edit the Producer class
        - To be *Runnable*
        - Implement *run()* method to push 20 int values to the stack and sleep a random amount of ms between insertions

4. Edit the Consumer class
    - To be *Runnable*
    - Implement *run()* method to pop 20 int values from the stack and sleep a random amount of ms between removals
5. The main method is fully implemented in the Factory class
    - Explore the way a stack, producer & consumer are instantiated
    - Examine how 4 different threads takes the producer and consumer and executes them simultaneously
6. Launch the application form *Factory* class
    - Examine the output
    - When threads gets blocked ?

Phase 2:

- In this phase you will transform the previous lab to work with *Executor* and *Callable* classes
- In order to complete this phase follow these steps:
    1. Leave *Stack* class unchanged
    2. *Producer* class
        - Turn it into a *Callable<Integer>* (means that the *call()* method returns an in value – 0=OK, 1=Failure)
        - Implement the *call()* method to act just like the *Producer.run()* method. In case of throwing exception – catch it and return 1. If everything works well return 0.
    3. *Consumer* class
        - Turn it into a *Callable<Integer>* (means that the *call()* method returns an in value – 0=OK, 1=Failure)
        - Implement the *call()* method to act just like the *Consumer.run()* method. In case of throwing exception – catch it and return 1. If everything works well return 0.

4. *Factory* class main method
    - Create a cached thread pool via *Executors* class
    - Instantiate 4 callable objects – 2 of type *Producer* & 2 of type *Consumer*
    - Create a type safe collection that hosts those *Callable* [*Collection<Callable<Integer>>*] and add all four instances to it
    - Use the cached thread pool executor to launch all the *Callable* that are in the collection. No needs to process return values.
    - Shutdown the executor so it will not accept any new tasks and exit once all *Callable* objects ends
5. Launch the application form *Factory* class
    - Examine the output
    - Replace the cached thread pool with fixed thread pool with 2 threads only
    - Launch the application again and examine the output

Lab – Effective Java

In this exercise you will create your own version that does exactly what an existing code does – but this time – much better.

The lab consists of *bad* and *good* packages. Both packages holds classes that should do the same business logic:

bad.BadImplementation

bad.Person

good.GoodImplementation

good.Person

Your job is to fill the *good.GoodImplementation* class so it will follow the same logic as implemented in *bad.BadImplementation* and change few things in *good.Person* class which is also currently a copy of the bad.Person class.

Both classes are located at *labs\lab 7\exercise*

- **Feature Envy , Inline, Calling methods within a loop –** Currently, occurs every time the private method *good.GoodImplementation.printData()* points to a Person instance and calls its *get()* methods. It means that :
    - the caller is tightly coupled with the *Person* class
    - a lot of methods are invoked within the loop in the *good.GoodImplementation.print()*

    1. Update the *good.Person* class to have a *toString()* method to return in a single line all that data
    2. Update *good.GoodImplementation.print()* to use *Person.toString()*

- **Intensive string concat -** *good.GoodImplementation.print()* uses string concat which causes a huge amount of objects to be generated during runtime
    1. Update - *good.GoodImplementation.print()* method to use *StringBuffer*
- Now, apply the required changes in order to enhance the *good.GoodImplementation.printYoungest()* method
- In order to view results and measure performance improvements, a test application is fully provided – *Test.class* that does the following for both bad and good implementations:
    - Creates and instantiates a Person array with the size of 20
    - Instantiates the implementation classes (*BadImplementation* & *GoodImplementation*) with the Person array
    - Keeps a start time
    - Invokes both *print()* & *printYoungest()* methods from the implementation classes a 100 times (!)
    - Calculates the duration of the operation in ms and prints it

  This is done for the bad implementation via *testBad()* method and for good implementation via *testGood()*. The *main()* method calls both so you can get a clear picture of the differences between the two.