Concurrent programming

Thread synchronization and coordination

There are different ways of synchronizing or coordinating threads when they are launched from the same application. We can, for instance, assign different priorities to each thread so that some of them are faster than the rest. We can also join threads, i.e., make a thread wait until another thread finishes its task completely. From that point on, there are more complex synchronization structures, such as mutual exclusion, locks... We will see some of this techniques in this section.

1. Basic coordination. Joining threads

If we want a thread to wait until another thread finishes, we can use the join method from the thread that we want to wait for. In this example, the main application creates a thread and waits until it finishes before going on:

```
public static void main(String[] args)
{
    Thread t = new MyThread();
    t.start();
    t.join();
}
```

In fact, `join` method can throw an `InterruptedException`, so we have to catch it:

```
public static void main(String[] args)
{
    Thread t = new MyThread();
    t.start();
    try
    {
        t.join();
    } catch (InterruptedException e) {
        ...
    }
}
```

If we want a secondary thread (not main program) to wait for another thread, then we need to tell this thread which is the thread it must wait for. We typically use an attribute inside thread class to store this information:

```
public class MyThread extends Thread
{
    Thread waitThread;

    // We will use this constructor
    // if thread does not have to wait for anyone
    public MyThread()
    {
        waitThread = null;
    }

    // We will use this constructor
    // if thread has to wait for thread "wt"
    public MyThread(Thread wt)
    {
        waitThread = wt;
    }

    // We check if waitThread attribute is not null,
    // and then call the join method before keep on running
    public void run()
    {
        if (waitThread != null)
            waitThread.join();
        ...
    }
}
```

Then, in main application, we create two threads of type MyThread, and ask one of them to wait for the other:

```
public static void main(String[] args)
{
    Thread t1 = new MyThread();
    Thread t2 = new MyThread(t1);
    t1.start();
    // We start thread t2, but it will not run until t1 finishes
    t2.start();
}
```

Note: When using `join` in secondary threads, make sure to handle potential `InterruptedException` to avoid unexpected behavior in the thread coordination.

```
public class MyThread extends Thread
{
    Thread waitThread;

    public MyThread(Thread wt)
    {
        waitThread = wt;
    }

    public void run()
    {
        try
        {
            if (waitThread != null)
                waitThread.join();
        } catch (InterruptedException e) {
            System.err.println("Thread interrupted");
        }
        ...
    }
}
```

Exercise 1:

Create a project called **ThreadRaceJoin** based on previous project of *Exercise 3*. Change the behavior of the three running threads (A, B and C) so that each one starts running when previous thread has finished:

- Thread A will start at the beginning of the program.
- Thread B will start when thread A finishes.
- Thread C will start when thread B finishes.
- Main program will wait until the last thread (C) finishes the race.

Exercise 2:

Create a project called **MultiplierThreadsJoin** based in previous project of *Exercise 2*. Change the behavior of the main application so that it waits for each thread to finish before starting the following. Therefore, all the multiplication tables will be shown in order:

```
0 x 0 = 0
0 x 1 = 0
...
0 x 10 = 0
1 x 0 = 0
...
```

2. Access to shared resources. The need of thread synchronization

It is quite usual that multiple threads want to get the same resource (e.g. a variable, a text file, a database...), and it is difficult to guarantee that the information in that resource will not be mistakenly modified (for instance, that a thread changes the value of a variable while another thread is using it).

The piece of code that is in charge of allowing threads to get that shared resource is commonly called **critical section**. This code must not be executed by more than one thread at the same time. To achieve this, Java offers some options.

Let's see the problem in depth with this example: first of all, we create an object of class `Counter`, that will be shared among threads:

```
public class Counter
{
    int value;

    public Counter(int value)
    {
        this.value = value;
    }

    public void increment()
    {
        value++;
    }

    public void decrement()
    {
        value--;
    }

    public int getValue()
    {
        return value;
    }
}
```

You can see that `Counter` class has only one attribute, *value*, which is the value that will be read and/or modified by the threads, by calling `increment` or `decrement` methods.

Then, we create two types of threads: one that will increment `Counter` value in a loop, and another one that will decrement it:

```
public static void main(String[] args)
{
    Counter c = new Counter(100);

    Thread tinc = new Thread(() -> {
        for (int i = 0; i < 100; i++)
        {
            c.increment();
            try
            {
                Thread.sleep(100);
            } catch (InterruptedException e) { }
        }
        System.out.println("Finishing inc. Final value = "+c.getValue());
    });

    Thread tdec = new Thread(() -> {
        for (int i = 0; i < 100; i++)
        {
            c.decrement();
            try
            {
                Thread.sleep(100);
            } catch (InterruptedException e) { }
        }
        System.out.println("Finishing dec. Final value = "+c.getValue());
    });
    tinc.start();
    tdec.start();
}
```

What will happen? If you try this example on your IDE, you will find out that the final value of `c` object is different every time you run the example. Sometimes it is 105, sometimes it is 97... but it should be always 100 (it starts with 100, and then one thread is expected to increment the value 100 times and the other thread is expected to decrement it 100 times as well).

Why can this happen? Well, it may occur that `tinc` gets into `increment` method and then the control goes to `tdec`, that gets into `decrement` method. Then, one of these operations (either `value++` or `value--`) will have no effect. For instance, if `tinc` reads the value 100 and tries to set it to 101 but then the control goes to `tdec` that reads the same value 100 (`tinc` has not changed it yet) and sets it to 99, then when the control comes back to `tinc`, it will set value to 101, and the decrement will have disappeared.

To solve this problem, Java offers some mechanisms that can be used by a thread to check if there is any other thread executing the critical section before getting into it. If so, then the thread trying to get into the critical section is suspended by the synchronization mechanism. If there is more than one

thread waiting for a thread to finish the critical section, as soon as it finishes, the JVM chooses one of the waiting threads (randomly) to execute it. Let's see how this mechanism works, and its variations.

2.1. Synchronizing methods

One of the most basic methods of synchronization in Java is the `synchronized` keyword. We can use it to control the access to a method, so that it becomes a critical section.

Java only allows the execution of one critical section on each object. If the method is static, then this critical section is independent from all the objects of that class. In other words, Java only allows the execution of one critical section per object, and one static critical section per class.

In previous example, if we just add the `synchronized` keyword to the `increment` and `decrement` methods of Counter class:

```
public class Counter
{
    int value;

    public Counter(int value)
    {
        this.value = value;
    }

    public synchronized void increment()
    {
        value++;
    }

    public synchronized void decrement()
    {
        value--;
    }

    public int getValue()
    {
        return value;
    }
}
```

and we run again the program, we will notice that it works perfectly now. Why? Well, if `tinc` goes into `increment` method, then `tdec` will not be able to enter `decrement` method, and vice versa, so we will not have the problem of incrementing and decrementing the value at the same time, because both threads are sharing the same `Counter` object, and only one thread can be running a *synchronized* method at the same time.

You will also notice that the program runs slower than before. This is one of the effects of synchronization, it penalizes the performance of the application.

Exercise 3:

Create a project called **BankAccountSynchronized** with these classes and methods:

- A `BankAccount` class with an attribute called `balance` that will store how much money is there in the account. Add a constructor to initialize the money in the account, and two methods `addMoney` and `takeOutMoney`, that will add or take out the amount passed as a parameter. Add a `getBalance` method as well, to retrieve the current balance of the account.

```
public BankAccount(int balance) { ... }
public void addMoney(int money) { ... }
public void takeOutMoney(int money) { ... }
public int getBalance() { ... }
```

- A `BankThreadSave` class with an attribute of type `BankAccount`. You can either extend `Thread` class or implement `Runnable` interface to do that class. In the `run` method, the thread will add 100€ to the bank account for 5 times, sleeping 100 ms between each operation.
- A `BankThreadSpend` class with an attribute of type `BankAccount`. You can either extend `Thread` class or implement `Runnable` interface to do that class. In the `run` method, the thread will take out 100€ from the bank account for 5 times, sleeping 100 ms between each operation.
- From main class, create a `BankAccount` object, and an array of 20 `BankThreadSave` and 20 `BankThreadSpend` objects, using all of them the same `BankAccount` object. Start them all and see how the bank account balance changes (print a message somewhere to show the balance after each operation).
- At this point, you should have noticed that your bank account does not work properly. Add the synchronization mechanisms that you consider to solve the problem.

2.2. Synchronizing objects

We can also apply the `synchronized` keyword to a given object in a piece of code, passing the object as a parameter, this way:

```
public void myMethod()
{
    int someValue;
    ...
    synchronized(this)
    {
        someValue++;
        System.out.println("Value changed: " + someValue);
    }
    ...
}
```

Then, when a thread A tries to execute the instructions inside this block, it will not be able to do it if another thread B is already executing a critical section affecting the object `this`. As soon as this thread B finishes the critical section, the other thread A will wake up and enter the critical section.

Of course, we can use any other object with `synchronized` keyword. For instance, if we have an object called `file` and we want to create a critical section around it, we can do it like this:

```
public void someMethod()
{
    ...
    synchronized(file)
    {
        ... // Critical section
    }
    ...
}
```

Note: When synchronizing on objects, ensure that the object used for synchronization (`file` in this case) is not exposed publicly to avoid potential synchronization issues caused by external threads.

> **Exercise 4:**
>
> Create a project **BankAccountSynchronizedObject** based on previous exercise. In this case, you can't synchronize any method, you can only synchronize objects. What changes would you add to the project to make sure that it will keep on running properly?

3. Thread priorities

When we have multiple threads running on a program, we can change the priority of each thread, so that those threads with higher priority will get the processor more frequently. This feature is only applied to threads, not to processes, since JVM is not responsible for outer processes.

Priorities in threads are just integers from 1 (stored in `Thread.MIN_PRIORITY` constant) to 10 (stored in `Thread.MAX_PRIORITY` constant). By default, every thread has a priority of 5 (`Thread.NORM_PRIORITY` constant), and every thread inherits its parent's priority, unless we change it later.

To change the priority of a thread, we can use its `setPriority` method, passing an integer as a parameter. We can also check the thread's priority with `getPriority`.

```
Thread t1 = new MyThread();
Thread t2 = new MyThread();
Thread t3 = new MyThread();

t1.setPriority(Thread.MIN_PRIORITY);
t2.setPriority(Thread.NORM_PRIORITY);
t3.setPriority(Thread.MAX_PRIORITY);

System.out.println("Priority of thread #2 is " + t2.getPriority());
```

3.1. Operating system dependency

There is a problem with priorities depending on the operating system that we are using. In Windows systems, you will see that your threads behave more or less according to their priorities, but in Linux and Mac OS X systems, the priority that we try to set to each thread has no effect. So we have to keep in mind that the expected behavior of our threads is not guaranteed, and it will depend on the operating system, unless we look for another options.

If we need to be sure that some threads will have a higher priority, we can't rely on `setPriority` method, because the operating system may ignore these priorities. An alternative option is to use random numbers and the `yield` or `sleep` methods to force threads to free the processor according to their real priority. Let's have a look at this example:

```
public class MyPrioritizedThread extends Thread
{
    int priority;

    public MyPrioritizedThread(int priority)
    {
        this.priority = priority;
    }

    @Override
    public void run()
    {
        java.util.Random r =
            new java.util.Random(System.currentTimeMillis());

        while (condition)
        {
            // Generate a random number between 1 and 10
            int number = r.nextInt(10) + 1;

            // If this number is greater or equal than thread's
            // priority, yield
            if (number >= priority)
                Thread.yield();

            ... // Rest of the instructions for our run loop
        }
    }
}
```

We define our `Thread` subclass with its own `priority` attribute, that will be managed by our code. In `run` method, we generate a random number between 1 (that will correspond to `Thread.MIN_PRIORITY`) and 10 (that will correspond to `Thread.MAX_PRIORITY`). If this number is greater or equal than `priority` attribute, then our thread will yield. Notice that threads with lower priorities (i.e. closer to 1) will yield more frequently, and threads with higher priorities (i.e. closer to 10) will yield from time to time.

If the task planner ignores the `yield` instruction and our priorities do not seem to have any effect, then replace the `yield` instruction with some sleeping time:

```
if (number >= priority)
    try
    {
        Thread.sleep(5);
    } catch (Exception e) {}
```

The more milliseconds you put your threads to sleep, the more time will take for threads with lower priorities to finish their task. It is up to you to adjust the most appropriate number of milliseconds, depending on the application you are developing.

> **Exercise 5:**
>
> Create a project called **ThreadRacePriorities** based in *ThreadRace* project of *Exercise 5*. Modify the code so that thread *A* has `MAX_PRIORITY`, thread *B* has `NORM_PRIORITY` and thread *C* has `MIN_PRIORITY`. Do it with `setPriority` method. Try to check or see the results in different operating systems.

> **Exercise 6:**
>
> Create a project called **ThreadRacePrioritiesRandom** that changes the assignment of priorities of previous exercise for the second option explained (random numbers and *yield/sleep* method). Try to check or see the results in different operating systems.

4. The producer-consumer problem

The producer-consumer problem is a classic problem in concurrent programming. In this type of problems, we have a data buffer, some producers that put data into that buffer and some consumers that take data from the buffer. We have to make sure that consumers will not try to take data when the buffer is empty and, in some cases, that producers will not produce more data until consumers take the existing one, or if buffer is full.

In these type of problems the use of `synchronized` keyword is not enough. We have to add some mechanisms to make either producers or consumers wait until the other side has done its job. To do this, we can use the `wait`, `notify` and `notifyAll` methods, from `Object` class:

- The `wait` method can be called inside a synchronized block. Then, the JVM puts the thread to sleep and releases the object controlled by this synchronized block, so that other threads running synchronized blocks of the same object can go on.
- The `notify` or `notifyAll` methods are called by a thread that has finished its task inside a critical section, before leaving it, to tell the JVM that it can wake up a thread previously put to sleep with a `wait` method. The main difference between these two methods (`notify` and `notifyAll`) is that, with `notify`, the JVM chooses one thread waiting (randomly), whereas with `notifyAll` the JVM wakes up every thread waiting, and the first who gets into the critical section is the one who goes on (the others will keep on waiting).

Let's see an example: we will create two types of threads: a `Producer` that will put some data (for instance, an integer) into a given object (we will call it `SharedData`), and a `Consumer` that will get this data.

Our `SharedData` class is this one:

```
public class SharedData
{
    int data;

    public int get()
    {
        return data;
    }

    public void put(int newData)
    {
        data = newData;
    }
}
```

Our `Producer` and `Consumer` threads are these ones:

```
public class Producer extends Thread
{
    SharedData data;

    public Producer(SharedData data)
    {
        this.data = data;
    }

    @Override
    public void run()
    {
        for (int i = 0; i < 50; i++)
        {
            data.put(i);
            System.out.println("Produced number " + i);
            try
            {
                Thread.sleep(10);
            } catch (Exception e) {}
        }
    }
}

public class Consumer extends Thread
{
    SharedData data;

    public Consumer(SharedData data)
    {
        this.data = data;
    }

    @Override
    public void run()
    {
        for (int i = 0; i < 50; i++)
        {
            int n = data.get();
            System.out.println("Consumed number " + n);
            try
            {
                Thread.sleep(10);
            } catch (Exception e) {}
        }
    }
}
```

The main application will create a `SharedData` object and a thread of each type, and will start both.

```
public static void main(String[] args)
{
    SharedData sd = new SharedData();
    Producer p = new Producer(sd);
    Consumer c = new Consumer(sd);
    p.start();
    c.start();
}
```

If we copy this example and see how it works, we will see something like this:

```
Consumed number 0
Produced number 0
Consumed number 0
Produced number 1
Consumed number 1
Produced number 2
Produced number 3
Consumed number 3
Produced number 4
Consumed number 4
Consumed number 4
```

See how, sometimes, the producer puts numbers too fast, and sometimes, the consumer gets numbers too fast as well, so that they are not coordinated (the consumer may read twice the same number, or the producer may put two consecutive numbers).

We could think that, if we just add the `synchronized` keyword to `get` and `put` methods from `SharedData` class, we would solve the problem:

```
public class SharedData
{
    int data;

    public synchronized int get()
    {
        return data;
    }

    public synchronized void put(int newData)
    {
        data = newData;
    }
}
```

However, if we run the program again, we may notice that it still fails:

```
Consumed number 0
Produced number 0
Consumed number 1
Produced number 1
Produced number 2
Consumed number 1
Produced number 3
Consumed number 3
Produced number 4
Consumed number 4
```

In fact, there are two problems that we need to solve. But let's start with the most important one: producer and consumer have to work coordinated: as soon as the producer puts a number, the consumer can get it, and the producer will not be able to produce more numbers until the consumer gets the previous ones.

To do this, we need to add some changes to our `SharedData` class. First of all, we need a flag that tells producers and consumers who goes next. It will depend on whether there is new data to be consumed (turn for the consumer) or not (turn for the producer).

```
public class SharedData
{
    int data;
    boolean available = false;

    public synchronized int get()
    {
        available = false;
        return data;
    }

    public synchronized void put(int newData)
    {
        data = newData;
        available = true;
    }
}
```

Besides, we need to make sure that `get` and `put` methods will be called alternatively. To do this, we need to use the boolean flag and the `wait` and `notify/notifyAll` methods, this way:

```
public class SharedData
{
    int data;
    boolean available = false;

    public synchronized int get()
    {
        if (!available)
            try
            {
                wait();
            } catch (Exception e) {}
        available = false;
        notify();
        return data;
    }

    public synchronized void put(int newData)
    {
        if (available)
            try
            {
                wait();
            } catch (Exception e) {}
        data = newData;
        available = true;
        notify();
    }
}
```

See how we use `wait` and `notify` methods. Regarding `get` method (called by the `Consumer`), if there is nothing available, we wait. Then, we get the number, set the flag to false again and notify the other thread. In the `put` method (called by the `Producer`), if there is something available, we wait until someone notifies us. Then, we set the new data, set the flag to *true* again and notify the other thread.

If both threads try to get to the critical section at the same time, `Consumer` will have to wait (*available* flag is set to *false* at the beginning), and `Producer` will set the first data to be consumed. From then on, they alternate in the critical section, consuming and producing new data each time.

```
Consumed number 0
Produced number 0
Produced number 1
Consumed number 1
Consumed number 2
Produced number 2
Produced number 3
Consumed number 3
Produced number 4
Consumed number 4
...
```

Note: Using `notifyAll` instead of `notify` can prevent potential deadlocks when multiple threads are waiting for the same condition. While `notify` wakes up one random waiting thread, `notifyAll` ensures that all waiting threads are notified, reducing the risk of starvation.

```
public synchronized int get()
{
    if (!available)
        try
        {
            wait();
        } catch (Exception e) {}
    available = false;
    notifyAll();
    return data;
}

public synchronized void put(int newData)
{
    if (available)
        try
        {
            wait();
        } catch (Exception e) {}
    data = newData;
    available = true;
    notifyAll();
}
```

Exercise 7:

Create a project called **DishWasher**. We are going to simulate a dish washing process at home, when someone wash the dishes and someone else dries them. Create the following classes:

- A `Dish` class with just an integer attribute: the dish number (to identify the different dishes).
- A `DishPile` class that will store up to 5 dishes. It will have a `wash` method that will put a *dish* in the pile (if there is space available), and a `dry` method that will take a *dish* from the pile (if there is any). Maybe you will need a `Dish` parameter in `wash` method, to add a *dish* to the *pile*.
- A `Washer` thread that will receive a number N as a parameter, and a `DishPile` object. In its `run` method, it will put (wash) N dishes in the pile, with a pause of 50ms between each dish.
- A `Dryer` thread that will receive a number N as a parameter, and a `DishPile` object. In its `run` method, it will take out (dry) N dishes from the pile, with a pause of 100 ms between each dish.
- The main class will create the `DishPile` object, and a thread of each type (`Washer` and `Dryer`). They will have to wash/dry 20 dishes coordinately, so that the output must be something like this:

```
Washed dish #1, total in pile: 1
Drying dish #1, total in pile: 0
Washed dish #2, total in pile: 1
Drying dish #2, total in pile: 0
Washed dish #3, total in pile: 1
Washed dish #4, total in pile: 2
Drying dish #4, total in pile: 1
Washed dish #5, total in pile: 2
Washed dish #6, total in pile: 3
Drying dish #6, total in pile: 2
Washed dish #7, total in pile: 3
...
```