# Code Assignment 02 - Concurrent Programming

## Exercise 3 : Finding links in an http document

In this exercise, you are going to change the behavior of program `Xurl` in a way such that it fetches every link found in the downloaded document. For today, links of interest are those which are correctly embedded in `A` tags:

1. of the form : `<A ... href="an_url" ...>` or `<A ... href='an_url' ...>`, where any sequence of `key="value"` or `key='value'` attributes and spaces may take the place of the ellipsis and where additional spaces may be added around `=`, see the specifications; as html is non case sensitive an `A` may also be typed as `a` and any letter of `href` may be capitalized,
2. where `an_url` is considered as valid URL by the `MyURL` constructor,
3. and where the protocol for this URL is `http`.

For this exercise, you only have to list the URL of the selected links, by printing them on `System.out`. To do this:

1. insert in `Xurl` a call to method `URLprocessing.parseDocument` at a convenient place, so to pass it a buffer containing the document to be scanned; the argument is expected to match abstract type `CharSequence` and it could be either a `String` or the result of `CharBuffer.wrap(buffer, ...)` where `buffer` may be a `char[]`. If you are using the given solution for `Xurl`, this is done and you simply have to uncomment the call to `URLprocessing.parseDocument`,
2. fill in the body of `URLprocessing.parseDocument` so that each identified url is passed in turn as the argument of `handler.takeUrl()`.

A skeleton for `URLprocessing.java` is available from the download page. As initially defined, `handler.takeUrl()` only prints its argument. So a call to `parseDocument` must screen the given document, find each link of interest and pass the corresponding URL to `handler.takeUrl()` which prints this URL, each on a line, and nothing else. You don't have yet to visit the discovered URLs. After some convincing tests, submit your file `URLprocessing.java` through the usual upload form. This **sample page** allows a small test.

## Exercise 4 : Iteratively downloading an http document and those directly and indirectly linked

The algorithm to be implemented is equivalent to the breadth-first traversal (aka Breadth-First Search) of a graph. As you should know, it uses both a FIFO queue, to store the URL to be explored, and a set, to store already known URLs (any kind of markers could alternatively be used).
The download page gives some useful material:

- interface `URLQueue` as the abstract definition of a queue,
- class `ListQueue` is given as a first implementation, to use for this exercise,
- skeleton file `Wget.java` for the new program, used to launch the whole.

So, you have to complete method `doIterative` of `Wget`, which will call `Xurl.query`, which calls `URLprocessing.parseDocument` and at last the new `handler` defined in `doIterative` is called for each url found. As a result, iterative calls to `Xurl` will download the set of linked pages, each only once. This **sample page** and those linked allow small tests. Draw the corresponding graph and verify that your program actually performs a BF traversal, whatever the initial URL is. This **website** may be used for a larger test (as the resulting structure is flattened, don't worry that some files are overwritten -- this isn't the point here).

Then, submit your `Wget.java` through the usual upload form.

## Exercise 5 : Multi-threaded download

Write in class `Wget` a method `doMultiThreaded(String requestedURL, String proxyHost, int proxyPort)`, inspired by `doIterative`, but where each call to `Xurl.query` is done in a new `Thread`. In others words, whenever an url comes out of the queue, an new `Thread` is launched with a call to `Xurl.query` as the body of its run method.
A first way to do this in Java, is to *extend* the `Thread` class. However, this way is discouraged because it gives two role to the same `Thread` object: the first is to define the task to be run (by extending `Thread`) and the second is to control the execution (through the already defined methods of `Thread`). Then a cleaner way is to *implement* the `Runnable` interface, to define the task to be run (calling `Xurl.query`), and to pass it as an argument to a `Thread`.

Now, some care is required to prevent race-conditions. First observe that calls to `handler.takeUrl` are executed by different threads. For an example, use `System.out.println(Thread.currentThread() + " was here.");`.
This first requires synchronization, such that the *Test-and-Set* operation on the `seen` structure is made atomic. Then, all operations on the queue must also be implemented in mutual exclusion. Devise a `SynchronizedListQueue` class for that. While the Java library provides some already synchronized data structures, you should not use them here, but you must explicitly implement the synchronization with the Java locking primitives. Note that your `SynchronizedListQueue` should have the same response as the given `ListQueue`, with an *infinite* capacity, and it should not block, but throw a `NoSuchElementException`, when called as the list is empty.

Also note that an empty queue no longer means the end of the job. As long as threads are still running, they can enqueue again some new URL. To be clean, when the `doMultiThreaded` function returns, all the threads you've lauched must have finished by their own. This means that they have gone to the end of their `run` method. You can manage the detection of the job completion with `Thread.activeCount()`, but don't make any particular assumptions on the number of active threads at the beginning of `doMultiThreaded`. You may simply consider that the number of such extra threads will not change during the execution of `doMultiThreaded`. It is also a good idea to let the main loop sleep a little time when the queue is empty.

When all is done and tested, submit both `SynchronizedListQueue.java` and `Wget.java` through the usual upload form.

## Exercise 6 : Multi-threaded download with a worker pool

Running some hundreds of "simultaneous" threads, with an active TCP connection for each, may be a huge load for your system, for the network and for the distant server. Then it is better to be limited to a reasonable number of threads. This is known as the *thread pool* pattern.

So, a fixed number of threads is launched and each has the task to loop on polling the queue for picking another URL to process. Write for that a method `doThreadedPool(int poolSize, String requestedURL, String proxyHost, int proxyPort)` in class `Wget`. Then, rather than busy-waiting on the queue, it is better to use a blocking queue where a call to `dequeue` will block as long as the queue is empty, until a call to `enqueue` (by another thread) allows to release. Again, you should not use some existing blocking data structures, but you must explicitly implement the feature with the Java `wait`/`notify` primitives. So, devise a `BlockingListQueue` class for that. Also note that a queue of fixed length, which blocks when it is full, makes no sense here. With such a queue, all worker threads could easily enter a deadlock.

Once the job completion has been detected, one must cleanly terminate all threads. Note that the `stop()` method of class `Thread` must NOT be used. It is deprecated since a very long date, for good reasons.
Then, the only safe way to terminate a thread is to ask it to terminate by itself (by returning from its `run` method), when it is in a consistent state. The idea is that one will implement the body of the `run` method as a big `while(...)` loop, so that, a thread which succeeds in dequeuing an url, even interrupted, will complete the turn to process this url before terminating. Here, two ways for signaling exist:

- **out-of-band signaling**, using the *interrupt* mechanism for implementing a `while(!Thread.interrupted())` loop. Carefully read the documentation of the different related calls of the Thread API and also take care of `InterruptedException`. This is a general purpose mechanism, that one must absolutely understand,
- **in-band signaling**, when a communication channel is in the game, here the queue, one can simply push a special symbol into the channel, being supposed that such a

symbol, different from any ordinary content, may be identified. Here, for instance, `**STOP**` may not be confused with an URL.

For interoperability, we will define a `BlockingListQueue` class which is compatible with both ways. That is, regardless of the way used to push the signal, a thread waiting to dequeue will be properly notified as it expects. So, a blocked thread which is awakened by an `InterruptedException`, must rearm its interrupt flag and also return `**STOP**` as the dequeued value. And, a thread dequeuing a `**STOP**` must also arm its interrupt flag.

First, work on a full out-of-band signaling. When all is done and tested, submit both `BlockingListQueue.java` and `Wget.java` through the usual upload form.

If you have some time left, you can then work on the implementation of in-band signaling, which should be easier.