

---

# **Concurrent Data Structures in Java**

**CSInParallel Project**

November 08, 2014

# CONTENTS

<b>1</b>	<b>A Single-process Web Crawler, or Spider</b>	<b>2</b>
1.1	To Start With . . . . .	2
<b>2</b>	<b>Improving the Spider</b>	<b>5</b>
2.1	First Question: How much work is there? . . . . .	5
2.2	Multiple Spiders to the rescue . . . . .	5
2.3	Can we share our original data structures? . . . . .	5
2.4	Guaranteed implementation: Use concurrent data structures . . . . .	6
<b>3</b>	<b>Using Multiple Processes</b>	<b>7</b>
3.1	Environment . . . . .	7
3.2	Executing sequentially (original non-threaded version) . . . . .	7
3.3	One spider . . . . .	7
3.4	Executing concurrently . . . . .	9
3.5	Multiple Spider ‘threads’ . . . . .	9
3.6	Key point: locking . . . . .	9
3.7	How to do this in Java . . . . .	9
3.8	Speedup . . . . .	10

**Prologue**

This lab activity asks you to complete some code provided so that it is able to crawl the web from a given starting URL. You will start by completing a sequential version of this code. Then you will work on a version that uses the libraries from `java.util.concurrent` to create a multi-threaded version.

**Prerequisites**

Some existing knowledge of the Java data structures `LinkedList` and `ArrayList` is necessary.

**Code**

Please download `ConcurrentDataStructures.jar`. Inside the `src` directory are two packages: one for the original sequential web crawler called `lab/spider`, and another for the threaded spider called `lab/concurrentSpider`. The first chapter linked below starts you out with the sequential spider using conventional Java data structures to hold the URLs it encounters. You will then continue on to create the threaded version, starting with this code provided.

---

# A SINGLE-PROCESS WEB CRAWLER, OR SPIDER

The World Wide web is aptly named when you consider the URL links found in pages. One page can have many links in it that take a viewer to another page, which has more links, and so on, forming a very large cyclic graph of interconnected pages.

In this lab you will be finishing some code for a web crawler, or spider, that will start with a 'seed' URL to a web page and read it to find links to other pages. Those links will be placed on a queue for further processing (we'll call this the **work** queue). When the initial page is processed, it is placed on another data structure to indicate that it has been visited already—this is the **finished** queue. This process is repeated for the next page whose link is on the work queue.

The code you will be given uses a Java library for parsing html files and looking for links (java.net.URL).

## 1.1 To Start With

Here are the files in the package lab.spider, which you will use as your starting point. Place these files into your Java IDE as indicated by your instructor.

```
AllWordsCounter.java    // contains a 'dictionary' to hold counts of how often a URL is encounterd
HttpHelper.java         // contains methods to read html pages and extract links; also can detect whe
RunSpider.java          // has main()
Spider.java             // the workhorse and the one you will be changing
TestHttpHelper.java     // JUnit test class
TestSpider.java         // JUnit test class
WordCount.java          // small helper class that holds a word and a count
```

The Spider.java class is the one that you should work on for this assignment. The RunSpider class contains main() and uses it. As the code stands now it doesn't really do anything if you run it.

Examine the code in the files. Begin by creating a class diagram that shows which classes 'use' or 'have' one of the other classes.

### 1.1.1 Java Collection Data Structures Used

Become familiar with the data structures that are used in this program. The diagram below illustrates them.

The class called *AllWordsCounter* is able to return an array of *WordCount* objects, each of which contains two data elements: a URL for a page, and a count of the number of times that URL was encountered by the crawler. In the **Spider** class, the method *getUrlCounts* will return this array, shown on the left in the picture below. This array can be used to examine what the spider encountered when it is finished running.

The **Spider** class also contains a Queue called **work** and a List called **finished**. The Spider will start at a given, predetermined page and read all the links to URLs on it (this is called scraping the page, which you will implement in the *processPage* method). The links it finds, illustrated on the far upper right in the following diagram, should be added to the **work** queue, and a counter for that link should be updated.

The **Spider** class will repeat this process in the *crawl* method by continuing to pull each URL to a page off the **work** queue, process it, and place it in the **finished** queue. This is done for a certain number of times as designated by the integer called *maxUrls*.

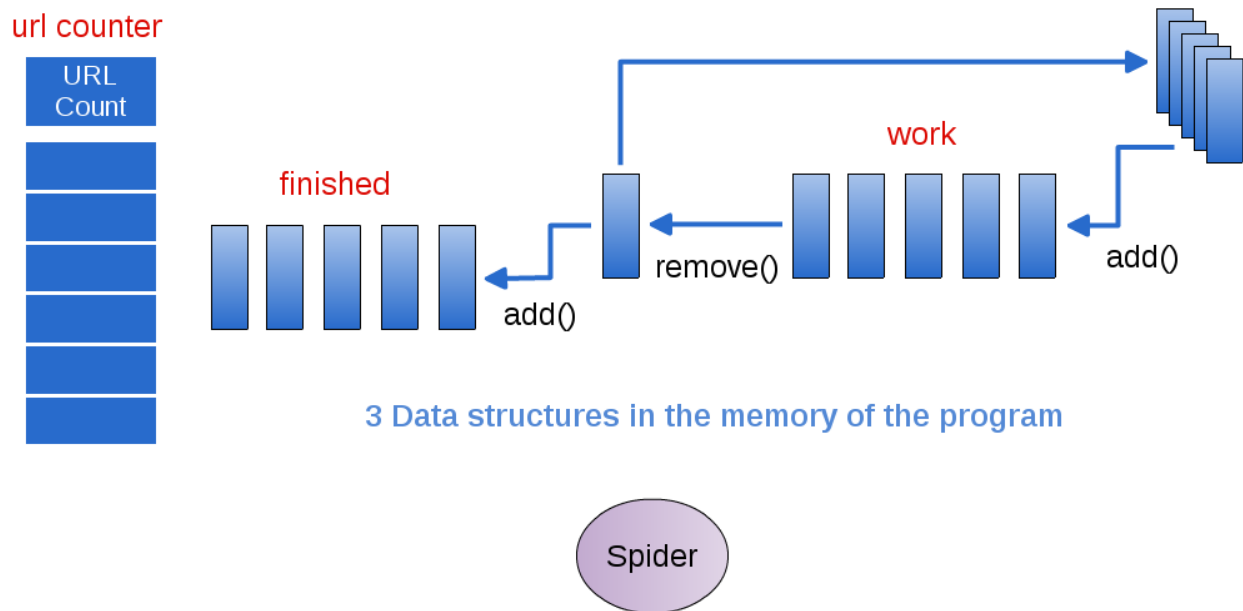


Figure 1.1: Single Spider accesses the data and does all the work

### 1.1.2 To Do

Your task is to finish the Spider class by doing the following:

- Complete the *processPage* method. When it works, one of the *TestSpider* unit tests should pass.
- Complete the *crawl()* method. When it works, both *TestSpider* unit tests should pass.

**Note:** There are comments in these methods to help assist you.

Once your unit tests pass, you should be able to run the code by executing the main method of the *RunSpider* class, which is currently 'hard-coded' to start at *macalester.edu*, and see it produce the URLs found when crawling, along with how many times it saw them.

**Try This:**

- Experiment with this variable found in Spider: `maxurls` If you double it, how many new urls were encountered? You might want to make a method that would answer this for you.
- Experiment with the `BEGNNING_URL` variable found in `RunSpider` by choosing some other pages of interest to you as starting points.

# IMPROVING THE SPIDER

## 2.1 First Question: How much work is there?

Once you have a completed working Spider, let's examine how much work it has to do. Try some experiments in which you continue using increasing values of `maxUrls` in the Spider. Please note that you can provide this value in its constructor.

**TO DO:** Add a method to the Spider that enables you to ask how many pages are still left to work on in the 'work' queue. You may also want to add a method to know how many pages have been finished.

**TO DO:** Change the `RunSpider` class to run some experiments with different values of `maxUrls` by executing several Spiders, one right after the other, with increasing numbers of URLs. For each value of `maxUrls`, report on how much work is left to do. How quickly is our single Spider overloaded with work?

## 2.2 Multiple Spiders to the rescue

Now let's examine how we can use multiple spiders working at the same time on this problem. Your instructor will take a moment to explain how we will use a technique called threads to run many spiders at the same time, each of who will access the work, finished, and `urlCounter` queue. Then you will try this out.

**Note:** more details about the implementation can be found on the next page, which you can get to by following 'next' in the upper and lower right of this page or going to the 'Next Topic' in the menu on the left.

There is a new `lab.concurrentSpider` package included in the code. Examine the **`RunThreadedSpider`** class. Note that we now use a Java class called a `Thread` to begin running multiple instances of the **`ConcurrentSpider`**, one each per `Thread` that is started. The Spider is now in this class called `ConcurrentSpider`, and implements an interface called `Runnable`.

A *key feature* of concurrently running Spiders is that they must share the same data structures in order to work together. To do this, we need to place the data structures they are working on in one class and create one instance of that class in `RunConcurrentSpider`. Then each new 'Runnable' `ConcurrentSpider` will receive a reference to that class of shared data structures. We provide a class called **`SharedSpiderData`** for this purpose.

## 2.3 Can we share our original data structures?

We could attempt to use the original `LinkedList` and `ArrayList` data structures and share those among the threads. However, these are not 'thread safe', that is they are not guaranteed to behave properly when multiple threads are accessing and updating them at the same time.

## 2.4 Guaranteed implementation: Use concurrent data structures

To ensure our code will work correctly using multiple threads, we will use the new Java Concurrent Data Structures from the package `java.util.concurrent`. Begin with the file `SharedSpiderData` to see the types of shared, thread-safe data structures we will use for this version of the multi-threaded crawler.

### 2.4.1 To Do

Finish the class called `ConcurrentSpider` so that it uses the new concurrent data structures when scraping the pages and keeping track of what has finished. You will need to discover what methods on the concurrent data structures (`ArrayBlockingQueue`, `ConcurrentLinkedQueue`) are available for adding and removing elements.

You will also find it useful to include the ability to have the `RunThreadedSpider` class be able to determine how much overall work was completed.

**Try This:**

- You can try using different numbers of threads, depending on how much your machine can handle, by changing the `NUM_THREADS` variable in the `RunThreadedSpider` class.
- Experiment with the parameter found in `ConcurrentSpider` constructor: `maxUrls`. If you double it, how many new urls were encountered? Now that you have all these spider threads, you can likely scrape more URLs.
- Experiment with the `BEGNNING_URL` variable found in `RunSpider` by choosing some other pages of interest to you as starting points.



# USING MULTIPLE PROCESSES

## 3.1 Environment

1. On Machines with multiple cores,
  - **Threads within a program can enable concurrency**
    - Run in parallel on multiple cores
  - Threads can share data in memory
  - **Ideally, needed work can get done faster**
    - **There is a speedup in the computation when using multiple threads as opposed to using 1 thread**
      - \* We measure speedup by taking the time to run the original non-threaded version and dividing it by the time to run the threaded version.
2. Threads are not a concept associated with clusters of machines, only multicore machines

## 3.2 Executing sequentially (original non-threaded version)

## 3.3 One spider

1. The original web crawling spider that you've been working on looks like this:
  - **3 data structures holding information**
    - Accessed individually by the Spider class
  - The Spider class does all the work, one step at a time
  - **In this case, the 'work' is piling up!**
    - Each page that is visited has many more links to follow
2. What might make this process faster so that more work gets done?

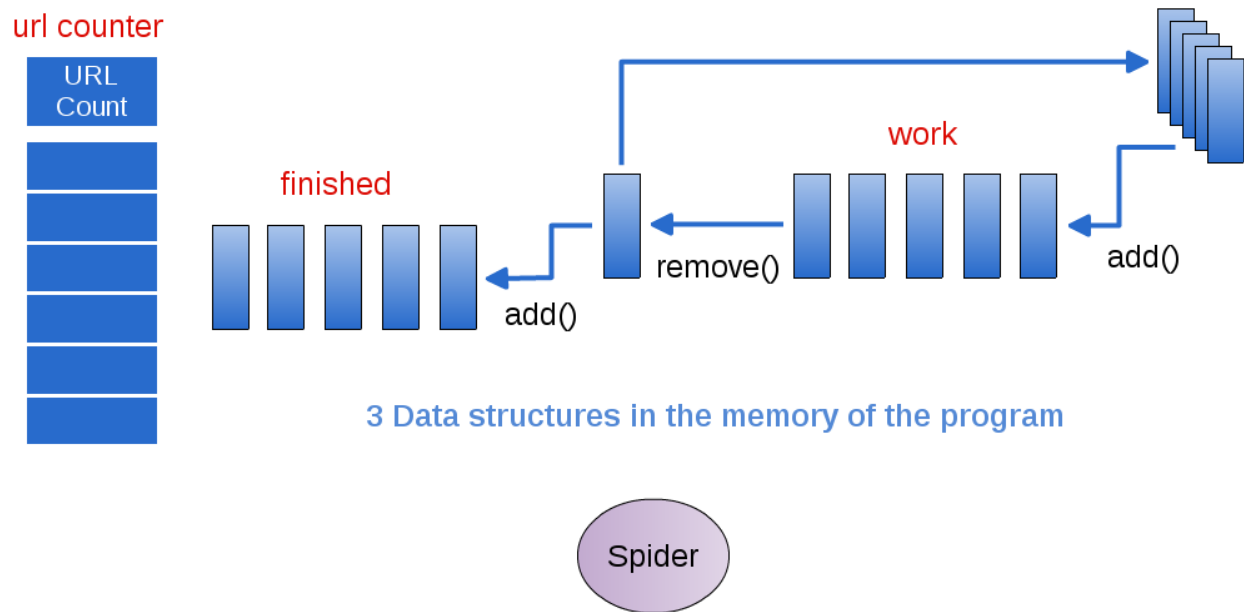


Figure 3.1: Single Spider accesses the data and does all the work

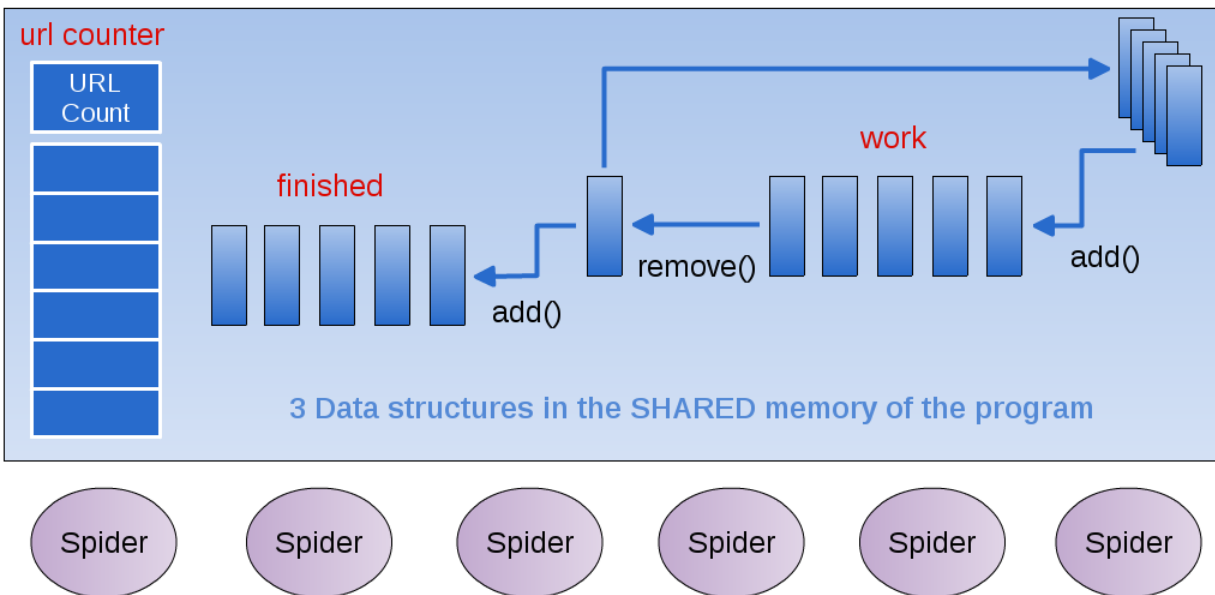


Figure 3.2: Multiple Spider 'Runnable' Threads all access the shared data

## 3.4 Executing concurrently

## 3.5 Multiple Spider ‘threads’

- **What happens when several threads need to read and write from the ‘work’ data structure?**
  - Imagine yourself as a spider working with a group of others
  - **What actions are involved when you:**
    - \* Grab a new page to work on from the work data structure
    - \* Save new links to the ‘work’ data structure
    - \* Store the completed page in ‘done’ data structure

## 3.6 Key point: locking

- **For certain operations on a shared data structure,**
  - Other threads must be barred from accessing when another thread is executing that operation
- These operations must be atomic: only one thread can be executing this operation at a time
- Which operations on the work data structure should be atomic?

## 3.7 How to do this in Java

### 3.7.1 Creating Threads in Java

- Build a class that implements the Runnable interface:

```
public interface Runnable {  
    abstract public void run();  
}
```

In our case, this is called ConcurrentSpider

- In a class containing ‘main’, create each new thread and pass it a new instance of the ‘Runnable’ ConcurrentSpider class

### 3.7.2 Sharing the Data

- Create your shared data structures in a separate class
- Create one instance of the shared data class in the ‘main’ class
- Pass that instance of the shared data to each instance of the ‘Runnable’ ConcurrentSpider class via the constructor

### 3.7.3 The Shared Data

- Java has special data structures designed to be shared by Threads
- See the documentation for [java.util.concurrent](#)
- **We're using:**
  - ArrayBlockingQueue
  - ConcurrentLinkedQueue
  - (ConcurrentHashMap, inside another class provided for you to hold the counts of the URLs)

## 3.8 Speedup

Speedup is a number we can compute by running experiments. It gives us an indication of how much faster our threaded version is than our original 'sequential' version with one Spider.

We measure speedup by taking the time to run the original non-threaded version and dividing it by the time to run the threaded version. If when using 2 threads on the same number of URLs the code takes half the amount of time, we would have perfect speedup of 2. This rarely happens, and we expect our speedup to be less than perfect due to some amount of overhead incurred when running threads that are updating shared data structures.

**Dig Deeper:**

Once you've implemented your solution,

1. Roughly determine the speedup of your threaded version
  - For varying numbers of threads, keeping the number of URLs processed constant
  - What will you need to measure?
2. Write a short report analyzing the speedup of your threaded solution