Assignment	Write a program that downloads a file from the Internet using multiple threads. The application has a Graphical Interface written in JavaFX, where the user can a) input the URL to download, b) select the destination file, c) view download progress and speed, and d) cancel a pending download. **80% credit** for an application that uses 1 background task (serial download). **100% credit** for an application that uses multiple background tasks to download parts of the file simultaneously. Upon completion, the download file must be an exact copy of the thing downloaded (no corruption).
What to Submit	1. Create a project named **`flashget`** and commit it to Github Classroom using the repository that will be created for you. Go to the URL and "accept" the assignment. 2. Create a *runnable JAR file* of your application with the name flashget.jar in the top level folder of your project.

Requirements

Write a graphical application that downloads a file using one or more threads, where each thread downloads one part of the file and all threads run simultaneously.

There is partial credit (shown above) for an application that downloads a URL using only one background task, and full credit for an application that uses multiple threads for simultaneously downloading parts of the URL.

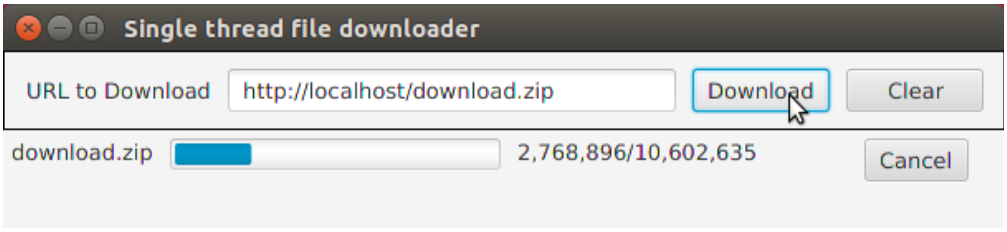
1. The threads should be implemented as JavaFX "Tasks" so that the the UI can be updated with the download progress, and the download can be cancelled.

2. The UI should show the download progress using both a ProgressBar and a count of the number of bytes downloaded (similar to downloading a file in Firefox or Chrome).

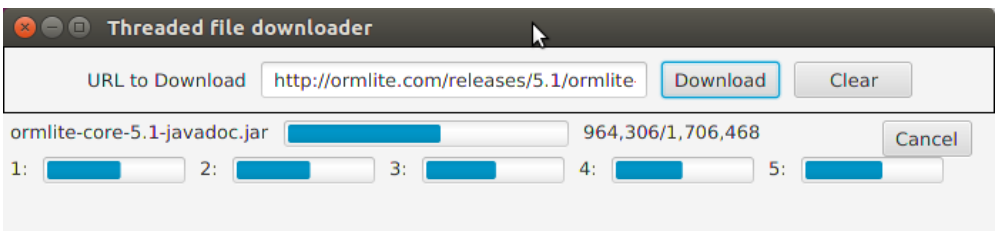
3. When the user clicks the "Download" button, open a FileChooser dialog to ask him/her to select the destination for download. FileChooser lets you specify an initial directory. Try to choose a good default for the download directory, and remember the user's previous choice so if he downloads another file you show a FileChooser with the same directory as previous download. You can get the user's home directory using System.getProperty("user.home"). From there, you can deduce the download directory.

4. User should be able to cancel a download at any time. The background tasks should stop.

5. The program should never crash, throw uncaught exceptions, or print on the console. When an exception occurs, catch it and display a message in the graphical user interface.



URL downloader using a single Task.



URL downloader using 5 background Tasks

Multi-Thread Downloader

This section describes a file downloader using multiple Tasks to download parts of the URL in parallel . The default should be to use 5 tasks (threads). For a small file you can use fewer threads.

Each thread downloads part of the file. The part sizes should be multiples of 4KB (4,096 bytes) or a larger multiple of 4KB, so that each output stream starts writing along sector or cluster boundaries of the storage media.

For example, to download a 400,000 byte file with a "chunk" size of 16KB, there are ceil(400000.0 / 16384) = 25 chunks. Using 5 background tasks (threads), each task would download:

thread1: download bytes 0 - 81,919 (80KB)

thread2: download bytes 81,920 - 153,839 (80KB)

thread3: download bytes 153,840 - 245,759 (80KB)

thread4: download bytes 245,760 - 327,679 (80KB)

thread5: download bytes 327,680 - 399,999 (72,320 bytes)

Below, you'll see how to download a part of a URL's content (by designating a range of bytes to download) and how to write to a file starting at any byte position using a *Random Access File*. Random access I/O is a part of almost all programming languages.

What Do You Need To Know?

To implement a solution to this, you need to know the following. Some items only apply to a multi-threaded downloader.

1. How to connect to a URL in Java
2. How to get the *size* of the content at a URL
3. How to read from a URL and save the result in a file.
4. How to download a part of a URL (not the whole content), by specifying a range of bytes
5. How to *write* to an arbitrary location in a local file (random access write) instead of from the beginning
6. How to design a JavaFX task for this.

Each of these is described below.

1. How to connect to a URL in Java?

Java has a java.net.URL class for accessing, reading, and sending information to URLs. To create a URL, use:

```
String urlname = "http://www.ku.ac.th/index.html";
URL url = null;
try {
     url = new URL( urlname );
} catch ( MalformedURLException e ) {
     System.err.println( e.getMessage() );
     return null;
}
```

2. How to you get the size of the file at a URL?

The URL class has a getConnection() method that lets you query and manipulate an HTTP connection. You must call getConnection() *before* opening an InputStream to read the url if you want to set any parameters on the connection.

Open a URL and query the content length.

```
long length = 0;
try {
    URL url = new URL( "http://someplace/somefile" );
    URLConnection connection = url.openConnection( );
    length = connection.getContentLengthLong( );
} catch (MalformedURLException ex) {
    // URL constructor may throw this
} catch (IOException ioe) {
    // getContentLengthLong may throw IOException
```

The URLConnection object has several query methods, including getContentLengthLong(). This method returns the length of the file or resource (if known) as a long. If you are certain that the file is smaller than 2GB you can also use getContentLength() which returns an int (but why bother?) If the size of the file or resource is not known, getContentLength() and getContentLengthLong() return -1.

3. How to you read a URL and write the data to a File?

After creating a URL object, querying the length, and optionally setting a request header (see next "howto" item), you can open an InputStream, and use it to read the URL content as bytes.

Hard disks and SSDs write data in sectors or clusters at a time; this is usually 4,096 bytes or a multiple of 4,096 (for SSD). HD and SSD are typically fastest at writing *many* sectors at a time, such as 256KB. Hence, its efficient to use an array (buffer) that is a multiple of 4KB (4,096) for writing.

```
final int BUFFERSIZE = 16*1024;
URL url = new URL( urlname );
URLConnection conn = url.openConnection();
InputStream in = conn.getInputStream( );
// suppose getOutputStream() is your method that creates
// a FileOutputStream from the File parameter
OutputStream out = getOutputStream( file );
byte[] buffer = new byte[BUFFERSIZE];
try {
    do {
        int n = in.read( buffer );
        if (n < 0) break;          // n < 0 means end of the input
        out.write(buffer, 0, n);  // write n bytes from buffer
        bytesRead += n;
    } while ( true );
}
catch( IOException ex ) {
    // handle it
}
finally {
    //TODO add try-catch to each close()
    in.close();
    out.close();
}
```

In the code above, in.read() returns an int that is the actual number of bytes read. This may be smaller than the buffer size, so don't assume that read() always fills the entire buffer with data. (InputStream also has a readFully method that does completely fill the byte array, if possible.)

The `finally` block is to ensure the connection to the URL is closed when done. You should also close the OutputStream to ensure all data is written to storage and free resources.

4. How do you download part of a URL from a given starting position?

To download different parts of the URL in parallel, we need a way to request a range of bytes to get from the URL. Then each download task can request a *different* range of bytes and save them to a file.

The HTTP 1.1 protocol allows a client to specify *request headers* that affect processing of the HTTP request. One request header is "`Range`" which requests the range of bytes to get. Here is an example HTTP "GET" request with a Range header set to get bytes 10,000 - 19,999. It also has two other headers.

```
GET /path/index.html HTTP/1.1
UserAgent: Mozilla
Host: www.example.com
Range: bytes=10000-19999
```

If you want to read from a given starting position (byte) to the end of the file, use the request header "`Range: bytes=10000-`" (no ending byte number).

In Java, you set HTTP request headers using `setRequestProperty( )` of the URLConnection object. Here is an example of setting the byte range 4096 - 8191 (bytes 4K - 8K).

```
int start = 4096;
int size = 4096;
URLConnection connection = url.openConnection();
String range = null;
if ( size > 0 ) {
    range = String.format("bytes=%d-%d", start, start+size-1);
}
else {
    // size not given, so read from start byte to end of file
    range = String.format("bytes=%d-", start);
}
connection.setRequestProperty("Range", range);

// now get the input stream for reading the part of the URL content
InputStream instream =  connection.getInputStream();
```

You must set request headers *before* calling `connection.getInputStream()`, as in the code above.

5. How do you write to an arbitrary position in a file?

Suppose you want to write to a file at a given location (`start`) rather than writing from the *beginning* of the file. Java has a RandomAccessFile class that lets you *seek* to any location in a file and then write (or read) starting at that location. The method is `seek(long)`. For example:

```
File file = new File( "output.txt" );
int start = 4098;   // start writing at byte number 4096 in file

// Create a random access file for synchronous output ("rwd" flags)
RandomAccessFile writer = new RandomAccessFile( file, "rwd" );
// seek to location (in bytes) to start writing to
```

```
writer.seek( start );
```

You can seek() to a location beyond the end of the file, too! Java will extend the file's size.

After seek(), you can write to the RandomAccessFile just like an ordinary OutputStream.

Since we only want to write a range of bytes, you might modify the do-while loop from the previous example.

```
RandomAccessFile writer = new RandomAccessFile( file, "rwd" );
// seek to location (in bytes) to start writing to
writer.seek( start );
byte [] buffer = new byte[BUFFERSIZE];
int bytesRead = 0;
try {
     do {
          int n = in.read( buffer );
          if ( n < 0 ) break;
          writer.write(buffer, 0, n);
          bytesRead += n;
     } while ( bytesRead < size );
} catch ( ... ) ...
```

For a multi-threaded downloader, each thread must open its own InputStream (using URLConnection) and its own output stream (using RandomAccessFile). It is OK for threads to simultaneously write to different parts of the same file using random access output. Be careful that each writer writes to a *different* part of the file and the locations (byte addresses) don't overlap.

6. How to Use Background Tasks to Download a URL

This is similar to the Counter using Worker Threads done in class during Week 11. The exercise and explanation are at: https://skeoop.github.io/javafx/WorkerThreads.html

You should to define a subclass of javafx.concurrent.Task for the work you want done using a worker thread, and write a call() method that performs the download, You will need attributes for information that the object needs to perform the download. An example single-threaded downloader is:

DownloadTask
-url: URL -outfile: File
DownloadTask(url, file) call( ): Long

You must have the UI show progress (bytes downloaded), so define DownloadTask as extending Task<Long>. The call() method downloads the file. Each time you read some data from the URL, call updateProgress() and updateValue(), similar to the Counter example done in class.

For a multi-threaded downloader, the DownloadTask will also need attributes for the start index and download size. You could set these via the constructor or add "set" methods.

DownloadTask
-url: URL -outfile: File -start: long -size: long
DownloadTask(url, file, start, size) call( ): Long

7. Managing Threads

For a multi-threaded downloader, consider using an *ExecutorService* to start the DownloadTask objects.

An *ExecutorService* manages a *pool* of threads, including reusing and scheduling threads. In this application we just need a fixed pool, and we want all the tasks to run simultaneously. If nthreads is the number of threads we want to use, then do:

```
ExecutorService executor = Executors.newFixedThreadPool( nthreads+1 );
Task<Long> task = new DownloadTask(...);
executor.execute( task );
```

Running too many threads at one time can slow down your application due to contention for CPU or I/O. You should probably limit the number of simultaneous threads to 4-10. Your app can guess at the number of threads to use. For smaller files, 1-2 threads may be faster than 5 threads.