Assignment	Write a program that downloads a URL using multiple threads to increase the download speed. Each thread downloads part of the URL using the http protocol. The application should have a graphical interface where the user can a) input the URL to download, b) view download progress and speed, and c) cancel a pending download.
What to Submit	1. Create a project named **`flashget`** and commit it to Github Classroom using the repository that will be created for you. The Github classroom assignment URL is being sent by enail. 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 threads, where each thread downloads one part of the file and all threads run simultaneously. The default should be 5 threads.

For example, to download a 100KB file using 5 threads, thread 1 would download bytes 0-20,479, thread 2 would download bytes 20,478-40,959, thread 3 would download bytes 40,960-61,439, etc. Java provides random access I/O, so (for example) thread 2 can *seek* to byte position 20KB in the output file and start writing there. All 5 threads can write to the same output file, provided they are careful not to write to same location. (See below for details.)

The user interface should display the progress of each thread and the overall download progress. The UI should remain responsible during download, and permit the user to cancel a pending download.

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

What Do You Need To Know?

To implement a solution to this, you need to know the following

1. How to connect to a URL in Java

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

3. How to download a part of a URL (not the whole file), by specifying a range of bytes

4. How to *write* to an arbitrary location in a local file

5. How to manage several threads as a group

Each of these is described below.

1. How to connect to a URL in Java?

Java has java.net.URL and java.net.URI classes with useful methods for using URLs, getting information about a URL, and reading a URL. To connect to a URL for reading, use:

```
String name = "http://www.ku.ac.th/index.html";
URL url = null;
try {
     URL url = new URL( name );
} catch ( MalformedURLException e ) {
     System.err.println( e.getMessage() );
     return null;
}
// get in input stream (may throw IOException)
InputStream instream = url.openStream( );
```

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 maniputate an HTTP connection. You must call this method *before* calling openStream if you want to set any parameters on the connection.

The URLConnection class has several query methods, including getContentLengthLong(). As you can guess, this method returns the length of the file or resource (if known). If you are certain that the file is smaller than 2GB you can also use getContentLength() which returns an int. If the size of the file or resource is not known, getContentLength() and getContentLengthLong() will return -1.

Open a URL and query the content length.

```
// use the code above to create the URL object
// but don't call openStream.
URL url = ...    // from previous code
long length = 0;
try {
     URLConnection connection = url.openConnection( );
     length = connection.getContentLengthLong( );
} catch (IOException ioe) {
     // handle it
}
```

3. How do you read part of a URL from a given starting position (a byte index)?

The HTTP 1.1 (and newer) protocol allows the client to specify many *request headers* that affect processing of the HTTP request. One of the headers is named "Range" which requests the range of bytes to get. Here is an example HTTP "GET" request with a Range header set to get bytes 1,000 - 1,999. It also has two other headers.

```
GET /path/index.html HTTP/1.1
UserAgent: Mozilla
Host: www.example.com
Range: bytes=1000-1999
```

If you want to read from a given starting position (byte) to the end of the file, use the header "Range: bytes=1000-" (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
     InputStream instream =  connection.getInputStream();
```

4. 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 Random Access File type that lets you *seek* to any location in a file and then write (or read) at that location. The method is `seek(long)`. For example:

```
File file = new File( "filename.txt" );
int start = 4098;   // start writing at this byte number in file
int size = 2048;
// for writing output at any location (random access)
RandomAccessFile writer = new RandomAccessFile( file, "rwd" );
                    // see Javadoc for meaning of "rwd" flags
writer.seek( start );
```

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

After executing the above statements to open a RandomAccessFile and seek to the place where we want to start writing, we can read data from the URLConnection (instream) and write to the output file.

```
// BUFFERSIZE is a constant for the size of each read.
// It probably should be a static constant, not a local variable.
final int BUFFERSIZE = 4096;
// size = number of bytes we want to read/write (from code above)
int buffsize = Math.min( size, BUFFERSIZE );
byte [] buffer = new byte[buffsize];
int bytesRead = 0;
try {
     do {
          int n = instream.read( buffer );
          if ( n < 0 ) break;
          writer.write(buffer, 0, n);
          bytesRead += n;
     } while ( bytesRead < size );
} catch (IOException e) { ... }
finally {
     try {
          if ( instream != null ) instream.close();
     } catch (IOException e) { /* ignore it */ }
}
```

Hard disks and SSDs are written in sectors or blocks at a time (not bytes). For a hard disk the sector size is typically 4,096 bytes. SSDs typically write whopping 256KB - 512KB at a time due to the way flash memory is addressed.

So for efficiently, try to perform reads and writes as multiples of 4096 (instead of, say, 4,000). Experiment with buffer sizes of *n**4096 (*n* = 1, 2, 4, 16, 64...) to find the size that maximizes download speed.

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

The `finally` block is to ensure the connection to the URL is closed when it is no longer needed. You should also close the `writer` to ensure all data is written to storage.

For a multi-threaded downloader, each thread will 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 file and the locations (byte addresses) don't overlap.

5. Managing Threads as a Group

In your application you may want to know when all the downloader threads have finished, or to cancel them all. There are a few ways to manage a collection of threads. The lowest level appreach is to create a ThreadGroup and add all the threads to the group. This is in the case you are creating and running threads yourself. ThreadGroup has an activeCount method the returns the number of active (live) threads in the group, and a method to interrupt all threads. (It is up to the threads to check for interrupts.)

A higher level way to use an *ExecutorService* in java.util.concurrent to create and manage threads. An *ExecutorService* manages a *pool* of threads, including reusing and scheduling threads. To create an ExecutorService, use a factory method of the Executors class. In this application we just need a fixed pool and will run all the threads simultaneously. If nthread is the number of threads we want to use, then do:

```
ExecutorService executor = Executors.newFixedThreadPool( nthread );
Runnable task = // create a file download task
executor.execute(task);
```

Running too many threads at one time will slow down your application due to contention for CPU use or I/O. You should probably limit the number of simultaneous threads to 4-10. To download a really huge file, also limit the chunk size for each thread. When a chunk finishes, use the executor to download another chunk (so you constantly have 4-10 threads downloading at one time).

6. Updating the UI

Flashget has a UI written in Swing or JavaFX that shows the download progress and when the download has finished. This requires some careful programming to prevent the UI from freezing (becoming unresponsive). Java states that updates to UI components should always be done on the *Event Dispatcher* thread, and long running tasks should be done on other threads (called *worker threads*) but not on the *Event Dispatcher* thread.

How do you manage this? Java has a class named SwingWorker that handles this for you. To use it in your application, create a subclass of SwingWorker and override the methods doInBackground, process(), and done( ). The doInBackground method is where you perform long-running work, so this method would start the other threads (or an object that starts the other threads) to download the file.

The *Java Tutorial* and my slides on *Threads in Swing* contain more detail and examples of using SwingWorker.