Implementation Scenario:

1. **First, create the head function** that sends a HEAD request to the server.
   The HEAD request returns meta-information about the resource:

1. Standard HTTP Headers

These headers are present almost always:

- **Content-Type** — the type of content (for example, `text/html`, `image/png`, `application/json`).
- **Content-Length** — the size of the resource in bytes.
- **Last-Modified** — the date of the last modification of the resource.
- **ETag** — a unique identifier for the version of the resource, used for caching.
- **Date** — the date and time when the server sent the response.
- **Server** — information about the server software (for example, `nginx/1.24`).
- **Connection** — the connection status (`keep-alive` or `close`).
- **Accept-Ranges** — support for partial requests (for example, `bytes`).

   From this, we will need **Content-Length** and **Accept-Ranges**.
   If Accept-Ranges equals "bytes", it means the server supports partial file downloads.

2. **Split the file into several parts.**
   Take Content-Length and divide it by 4 to get the byte ranges.
   (4 is chosen as the minimum for a modern processor with 2 cores / 4 threads)

3. **Form a list of ranges**, for example:

val ranges = listOf(0..1999, 2000..3999, 4000..5999, 6000..7999, 8000..9999)

4. **Start downloading chunks in parallel.**
   Use mapIndexed for parallel processing.

5. **For each range, send a GET request with the Range header.**
   For example: Range: bytes=1000-1999 — meaning “give me bytes from 1000 to 1999 inclusive.”

6. **Store the downloaded data in the results list by range index:**

results[index] = downloaded data of this range

7. **Assemble the file:**
   After all parts are downloaded and stored in results, write them sequentially to the final file.
   (This ensures correct order of writing)

Summary:

Split the file into ranges (splitToRanges), downloaded all ranges in parallel via downloadAllChunks, stored each chunk in results[index].
After all threads finished — wrote all chunks to the file via combineTheParts.

Main problem:

All parts of the file are in memory simultaneously.
For large files (1 GB, 2 GB or more), this can exhaust RAM or even crash the program.

Improved file download plan:

1. **Request file information**
    - Send a HEAD request to the server
    - Get the file size and check if the server supports partial download (Range)
2. **Create an empty file**
    - Create a file on disk to write data into later
3. **Split the file into parts**
    - Divide the file into multiple parts, each of size partSizeBytes (e.g., 10 MB)
4. **Split each part into chunks**
    - Divide each part into smaller chunks for parallel downloading
5. **Download chunks in parallel**
    - Create a thread for each chunk
    - Send an HTTP request with Range; the server returns only that piece
6. **Assemble parts into the file**
    - Chunks are written to their respective positions in the file
    - After all chunks are downloaded, parts are combined into the original file
7. **Repeat for all parts**
    - Continue until the entire file is fully downloaded
8. **File ready**
    - The main function prints a message about successful download

Unit tests for: getFileInfo, splitToRanges, and downloadChunk.

- splitToRanges — test for a single range and uneven splitting

  Mockito tests:
  When the function depends on external objects (e.g., the server):

- getFileInfo — function correctly reads both headers and handles when the server does not support ranges
- downloadChunk — test for response status other than 206 (Partial Content) and empty response body

Integration test: FileDownloadIntegrationTest

1. Verify the **full file download process**: getting file info, splitting into parts/chunks, parallel downloading, and writing to disk.
2. Compare the downloaded file with the original (using SHA-256).
3. Use a **real server** to test end-to-end functionality.


THANKS FOR READING!