

This lab has **five (5)** checkpoints.

## Learning Outcomes

Upon completing this lab, students should be able to

- **Trace** character handling code.
- Use Java New I/O `SeekableByteChannel` to read/write binary files.

## Introduction

1. Dr Ladd says you should know *three* things about the ASCII sequence (an none of them is any actual character code). List them:

### Solution:

1. The digits, character codes '0'-'9', are *sequential*.
2. The UPPERCASE alphabet, character codes 'A'-'Z', are *sequential*.
3. The lowercase alphabet, character codes 'a'-'z', are *sequential*.

2. What is the output of the following code?

```
String theString = "1 fish, 2 fish, 39 fish";
int countA = 0;
int countB = 0;
for (int i = 0; i < theString.length(); i++) {
    char ch = theString.charAt(i);
    if (('0' <= ch) && (ch <= '9')) countA++;
    if (ch != ' ') countB++;
}
System.out.printf("countA = %d, countB = %d\n", countA, countB);
```

✓ Show your solutions to these questions to the lab instructor. Be prepared to explain where they came from.

### Solution:

countA = 4, countB = 18

Counts digits in A and non-spaces in B.

3. Look in the `src` folder. There you will find `RB.java` (just a shell, actually) and `build.gradle`. Create a new Gradle project that is a Java application using Groovy as the build language.

Get rid of the `test` subdirectory and all `main/java` packages except for `binfile` (where all the files will go).

Put `src/RB.java` in the `binfile` package and replace `app/build.gradle` with `src/build.gradle`. The project should *build* and *run*.

Notice two things: `RB.java` must be provided with the name of a file (that would be using `--args` on the `gradle run` command) and, if you look in `build.gradle`, it is possible, using `-P"mainClass=..."`, to run different Java classes that contain `main`. By default, `binfile.RB` is run.

After making sure the number of parameters is correct, `RP.java` copies the file name into an object field. This is so that you can use it in the `run` method. New executables may require different numbers of command-line arguments.

✓ Make the project *build* and *run* under Gradle. Make sure it runs with a usage message when there are no arguments and does nothing when there is. Show your building/running code to the lab instructor.

4. Modify `RP.java` so that `run` opens a `SeekableByteChannel` on the named file (if possible), reads it *one byte* at a time, counts **UPPERCASE** letters in the input stream, and prints each byte (as a character) to standard output.

Java NIO uses different classes than you are used to to interact with the file system. Sample code will try to note the required **import** files at the top of the sample; if any are missing, compiler errors should be specific enough to find the full class name in the Java documentation.

- (a) In `run`: convert the file name (`String`) into a `Path`:

```
import java.nio.file.Path;
import java.nio.file.Paths;
...
void run() {
    Path path = Paths.get(fileName); // get a Path based on the full file name

    // OPEN and READ bytes
}
```

- (b) Use a *resource-based try-catch* to open a read/write `SeekableByteChannel` (the portal to the underlying file).

```
import java.io.IOException;
import java.nio.channels.SeekableByteChannel;
import java.nio.file.Files;
import static java.nio.file.StandardOpenOption.*; // WRITE and READ
...
// Replace OPEN and READ bytes
try (SeekableByteChannel channel = Files.newByteChannel(path, WRITE, READ)) {
    // the GOOD PATH!
} catch (IOException e) {
    // the BAD PATH!
}
```

- (c) Replace the `BAD PATH` with a print to `System.err` that says there was a problem reading the named input file (give the name in the error message) and uses `System.exit` with a non-zero return code to stop the program.

Test that this works by naming a non-existent file.

- (d) Replace the `GOOD PATH` with the following code. Open up a browser page on the Java documentation for the `ByteBuffer` class and take a look at `clear()` and `flip()`.

```
import java.nio.ByteBuffer;
...
ByteBuffer buffer = ByteBuffer.allocate(1); // get a 1 byte buffer

while (channel.read(buffer) > 0) { // gives bytes read or -1 on EOF
    buffer.flip(); // make it possible to GET what is in the buffer
    // set capacity to position (1), then position to 0
    byte b = buffer.get(); // read the first (only) byte in the buffer
    char ch = (char) b;    // convert to a character to avoid printing CODE

    // COUNT UPPERCASE CHARACTERS
    System.out.print(ch);

    buffer.clear(); // reset position to 0, capacity to 1 so read works
}
```

- (e) Test out the program by naming a source file (or the constitution file in `data`) and see that it echoes the file to the screen.

✓ Show the lab instructor the documentation for `ByteBuffer`. Be prepared to discuss the *position* and *capacity* fields of that class. Show the working code.

5. You need a counter variable, declared outside the `while` loop above, initialized to zero, and then printed *after* the loop finishes, displaying the number of uppercase letters seen.

Use an `if` statement to check whether or not `ch` is an uppercase letter. You can do that by comparing it to `'A'` and `'Z'` and using the fact that you should remember that the ASCII codes for the uppercase letters are **sequential**. So, if the `ch` value is *inside* the range of the uppercase, increment your count.

✓ Show the lab instructor the modified `RB.java` that counts and prints the count of uppercase letters in the code read.

6. Make a copy of the *read byte* program into a new Java file for an *xor byte* file that will read and write each byte in an input file, XORing it with a key letter.

Call the new program `XB.java`.

Get `XB` to compile. Use `-PMainClass=` to run `XB.java` rather than `RB.java`. You may want to have it print its name on the screen so you know the right one is running.

Adjust `XB.java` so that it expects *two* command-line arguments: the file name and a key letter. This means changing `main` to get the right count and updating the usage message if it is wrong.

You'll also want to update the constructor to take and save the `char` and update the call to the constructor.

Now, get it to compile.

The rest of the modification is all inside the `while` not end of file loop. The basics (ignoring flipping the buffer) is

```
get the byte
xor the byte with the key
put the byte
back file position up one place and write the buffer
```

Reading/writing the buffer updates the current position in the file by adding the actual number of bytes read or written. So, read one, backup one, write one and we have moved forward after overwriting the next byte.

The resulting `while` loop looks like this:

```
while (channel.read(buffer) > 0) {
    buffer.flip();           // make get work
    byte b = buffer.get();   // get byte out
    b = (byte) (b ^ key);    // XOR with key character

    buffer.clear();          // ready for put
    buffer.put(b);           // ready for get (write)
    buffer.flip();

    channel.position(channel.position() - 1); // backup
    channel.write(buffer);    // write
    buffer.clear();          // ready for put (read)
}
```

**IMPORTANT** Note that this code *changes* the file it is working on. You may want to make a copy of a file and then play with that copy so that if things do not go right you lose no work. The overwriting of the file is done *silently*.

When you run the code on a given file with a given letter, you can use the `less` utility program to print the contents of the file to the screen (or your `RB.java` program, if you like); `less` may ask if you want to see it even if it is a binary file (you do). The readable file is now not a plain text file.

What happens if you run the code on the same file *twice* with the same key letter?

✓ Show your working code to the lab instructor. Be prepared to show a file's contents, xor it, show it again, and then xor it back to the original.

7. [Not a separate checkpoint but necessary] Add and commit your full `gradle` project (if you `git add` just the root directory, `., git` will add all subdirectories recursively).

Create a repository in your Gitea turn-in organization, associate it with your lab work, and push the repo. This will be done at the end of every lab.