Panbyte

By Veronika Hanulíková (492760), Adam Hlaváček (493310), Lubomír Hrbáček (493077),
project repository

1 Overall project design

The main design goal of the project was to create an easily extensible program with a high focus on memory-safe operations. The functionality of the program is to be assured with both unit (manually coded) and integration tests (taken from the assignment).

1.1 Integration tests

Input and output examples from the assignment were adapted into a simple Python script which first builds the project using maven commands and then checks if the expected output is the same as the actual output. As these simple tests are written as simple Python, creating a new test case is as simple as adding a new line to the correct place in the code. These tests are executed after each push as a GitHub action.

1.2 Input processing

Our goal for a memory-safe program is to never store the whole input in memory (unless necessary), but rather read the input in chunks while outputting data as soon as it can be correctly formatted. To help with reading the input, a `VirtualByteReader` class was created. This class internally reads input by 4096B chunks whenever necessary, but provides interface for reading the input byte-by-byte. Furthermore, because the virtually read, but not-yet-requested raw input bytes are stored internally in memory, a helper method was created which can check the top of the byte queue if an arbitrary-long delimiter follows. With these two functions, we can then precisely detect when one input ends, and another begins with ease.

1.3 The idea behind parsers and formatters

The primary idea of the software is that everything processed is just bytes, which are only represented in different forms. For this, two main abstract classes exist -- `PanbyteInput` for parsing input & representing it internally as bytes and `PanbyteOutput` for formatting internal bytes into the requested output format. This makes implementing each input or output mode as easy as creating a child of one of these two classes.

The overall process of processing IO operation can be imagined as follows:

```
input bytes -> VirtualByteReader.readChunk() -> VirtualByteReader.readByte() ->
PanbyteInput.parse() -> PanbyteInput.flush() -> PanbyteOutput.format() ->
PanbyteOutput.sendDataAndFlush() -> output bytes
```

All inputs and outputs also check for the correctness and completeness of the input, namely if no disallowed characters are present and that everything could be parsed when end of file is reached.

2 Argument parser

All arguments are passed to the program from the command line. The `ArgumentParser` and `ArgumentValidator` classes are used to parse and validate the input arguments. The resulting options are stored in an instance of the `ProgramArguments` class.

2.1 Clarification on repeat options:

- repeating formats is considered an error
- repeating input and output file settings is considered an error
- repeating a delimiter is not considered an error, the last set is used
- repeating the input and output options is not considered an error, the last set is used

2.2 Representation of program arguments

The `ProgramArgument` class provides a simple interface to use custom arguments passed to the program from the command line. When an instance is created, it should guarantee that the given combination of arguments and options is valid and does not need to be checked further in the program.

Input and output formats can only take the values `bytes, bits, int, hex, array` as specified in the specification. Input and output options are divided into variables representing options for input (padding for bits format, endianness for int format), output (endianness for int format, number expression for array) and the bracket type (only for the array as output format).

2.3 Parsing of command line arguments

The `ArgumentParser` static class contains a simple algorithm for parsing input arguments.

First, the arguments are split into name and value pairs and stored in objects of class `Argument`. An array of strings is traversed in turn, and long and short switches are

distinguished. For short options, the next item in the array is stored as the argument value. For long options, the currently processed string is split.

Then the array of `Argument` objects is converted to the values stored in the `ProgramArguments` object. If given by the input, missing values are set to default.

2.4 Validation of arguments

Finally, after parsing all arguments, the values set in the `ProgramArguments` class object are validated. The combinations of input arguments and options and output arguments, options and any parentheses (for array format) are checked.

2.5 Encountered problems

Most of the problems were related to finding the ideal simple representation of program arguments and then validating all possible combinations for different formats.

3 Bytes parser

The implementation is done in `PanbyteRawInput` and `PanbyteRawOutput` classes. Because our project internally represents the data as raw bytes, the implementation is really simple in the form of pass-through.

3.1 Encountered problems

Because the bytes format should not consider a delimiter unless explicitly specified, we needed to modify the argument parser so that it records if a delimiter was specified. Otherwise, this mode would skip all `\n` characters as it has considered it as a default delimiter.

4 Parsing of hex format

The implementation is in the `PanbyteHexInput` and `PanbyteHexOutput` classes.

4.1 Hex input

Parsing is done in pairs of bytes, which are converted to and stored as a single byte. If only an odd number of bytes is provided, the class waits for more input to be read.

4.2 Hex output

Hex output is simple, as all that is needed to do is to format every internally stored byte into two bytes, which are the ASCII hex representation of the said byte.

5 Parsing of int format

The implementation is in the `PanbyteIntInput` and `PanbyteIntOutput` classes.

5.1 Int input

The parsing of integer from digits has to be divided into two main parts. Digits have to be accumulated into some buffer and processed only when there are no more digits on input because the bit representation of an integer depends on the whole integer, not on individual digits.

An instance of `BigInteger` class is created at the start of parsing. This class can represent arbitrarily large integers. For each digit from input, the internal `BigInteger` instance is multiplied by ten and the digit is converted to the new `BigInteger` instance and added to the internal one. This procedure allows us to process each digit separately, not storing arrays or Strings in memory but only one instance of `BigInteger` class.

When all input is read the internal `BigInteger` instance is converted to the byte array interpretation of the read integer. The byte array is reversed if there is the little-endian option selected - if not, byte array stays as it is, and it is passed on to the output parser.

5.2 Int output

The output procedure fully utilises the possibilities of the `BigInteger` class. Firstly it reverses the internal byte array if the little-endian option for output is selected. Then the `BigInteger` class is constructed from the internal byte array, and it is converted to the decimal `String` representation of the integer by method of `BigInteger` class. The resulting `String` is then outputted.

5.3 Encountered problems

As this parser was created after raw, bit and hex parsers, the structure of the program was ready for parsers that parses a chunk of data, outputs them and parses another chunk. That is not the case of this parser, so the structure had to be changed a little.

6 Parsing of bit format

The implementation is in the `PanbyteBitInput` and `PanbyteBitOutput` classes.

6.1 Bit input

Parsing is done in eighths of bytes, which are converted to bit values and stored as a single byte. If it is not possible to take the entire octet from the input, the next input is waited for.

Because of padding, it is necessary to distinguish when the parsed bytes are sent to the output. If padding is set to true, then the parsed eights can be sent to the output immediately. In the case of left padding, it is necessary to wait for the entire input to be read before adjusting the bytes to match the correct padding.

6.2 Bit output

The bit output is quite simple. To implement it, it is only necessary to read individual bits in parsed bytes and convert them to output values 1 and 0.

6.3 Encountered problems

One of the main problems in the implementation was the use of the `byte` type, which is signed in Java. Therefore, it was necessary to create a custom helper functions to implement the bit shift.