6. homework assignment; part 2, JAVA, Academic year 2013/2014; FER

Problem 5.

Write a command line program `MyShell` and put it in package `hr.fer.zemris.java.tecaj.hw6.shell`. When started, your program should write a greeting message to user (`Welcome to MyShell v 1.0`), write a prompt symbol and wait for the user to enter a command. The command can span across multiple lines. However, each line that is not the last line of command must end with a special symbol that is used to inform the shell that more lines as expected. We will refer to these symbols as `PROMPTSYMBOL` and `MORELINESSYMBOL`. For each line that is part of multi-line command (except for the first one) a shell must write `MULTILINESYMBOL` at the beginning. Your shell must provide a command symbol that can be used to change these symbols. See example:

```
C:\Users> java -cp bin  hr.fer.zemris.java.tecaj.hw6.shell.MyShell
Welcome to MyShell v 1.0
> symbol PROMPT
Symbol for PROMPT is '>'
> symbol PROMPT #
Symbol for PROMPT changed from '>' to '#'
# symbol \
| MORELINES \
| !
Symbol for MORELINES changed from '\' to '!'
# symbol !
| MORELINES
Symbol for MORELINES is '!'
# symbol MULTILINE
Symbol for MULTILINE is '|'
# exit
C:\Users>
```

In order to make your shell usable, you must provide following built-in commands: `charsets`, `cat`, `ls`, `tree`, `copy`, `mkdir`, `hexdump`.

Command `charsets` takes no arguments and lists names of supported charsets for your Java platform (see `Charset.availableCharsets()`). A single charset name is written per line.

Command `cat` takes one or two arguments. The first argument is path to some file and is mandatory. The second argument is charset name that should be used to interpret chars from bytes. If not provided, a default platform charset should be used (see `java.nio.charset.Charset` class for details). This command opens given file and writes its content to console.

Command `ls` takes a single argument – directory – and writes a directory listing (not recursive). Output should be formatted as in following example.

```
-rw-      53412 2009-03-15 12:59:31 azuriraj.ZIP
drwx       4096 2011-06-08 12:59:31 b
drwx       4096 2011-09-19 12:59:31 backup
-rw-   17345597 2009-02-18 12:59:31 backup-ferko-20090218.tgz
drwx       4096 2008-11-09 12:59:31 beskonacno
drwx       4096 2010-10-29 12:59:31 bin
-rwx        282 2011-02-10 12:59:31 burza.sh
-rwx        281 2011-02-10 12:59:31 burza.sh~
```

```
-rwx       1316 2009-09-10 12:59:31 burza_stat.sh
drwx       4096 2011-09-02 12:59:31 ca
drwx       4096 2008-09-02 12:59:31 CA
-rw-          0 2008-09-02 12:59:31 ca.key
```

The output consists of 4 columns. First column indicates if current object is directory (`d`), readable (`r`), writable (`w`) and executable (`x`). Second column contains object size in bytes that is right aligned and occupies 10 characters. Follows file creation date/time and finally file name.

To obtain file attributes (such as creation date/time), see the following snippet.

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
Path path = Paths.get("d:/tmp/javaPrimjeri/readme.txt");
BasicFileAttributeView faView = Files.getFileAttributeView(
      path, BasicFileAttributeView.class, LinkOption.NOFOLLOW_LINKS
);
BasicFileAttributes attributes = faView.readAttributes();
FileTime fileTime = attributes.creationTime();
String formattedDateTime = sdf.format(new Date(fileTime.toMillis()));
System.out.println(formattedDateTime);
```

The `tree` command expects a single argument: directory name and generates prints a tree (just as you did in lecture class, use same formatting).

The `copy` command expects two arguments: source file name and destination file name (i.e. paths and names). Is destination file exists, you should ask user is it allowed to overwrite it. Your `copy` command must work only with files (no directories). If the second argument is directory, you should assume that user wants to copy the original file in that directory using the original file name. You must implement copying yourself: you are not allowed to simply call copy methods from `Files` class.

The `mkdir` command takes a single argument: directory name, and creates the appropriate directory structure.

Finally, the `hexdump` command expects a single argument: file name, and produces hex-output as illustrated below.

```
00000000: 31 2E 20 4F 62 6A 65 63|74 53 74 61 63 6B 20 69 | 1. ObjectStack i
00000010: 6D 70 6C 65 6D 65 6E 74|61 63 69 6A 61 0D 0A 32 | mplementacija..2
00000020: 2E 20 4D 6F 64 65 6C 2D|4C 69 73 74 65 6E 65 72 | . Model-Listener
00000030: 20 69 6D 70 6C 65 6D 65|6E 74 61 63 69 6A 61 0D |  implementacija.
00000040: 0A                     |                       | .
```

In right part of output please replace all bytes whose value is less than 32 or greater than 127 with '.'.

If user provides invalid or wrong argument for any of commands (i.e. user provides directory name for `hexdump` command), appropriate message should be written and your shell should be prepared to accept a new command from user. Shell terminates when user gives `exit` command.

How should you organize your code? Define an interface `ShellCommand` that has a single method:

```
public ShellStatus executeCommand(BufferedReader in, BufferedWriter out, String[]
arguments)
```

which expects that `in` is wrapped stdin, that out is wrapped `stdout` and that `arguments` is an array of user-

provided arguments. `ShellStatus` should be enumeration `{CONTINUE, TERMINATE}`.

Implement each shell command as a class that implements `ShellCommand` interface. During shell startup, build a map of supported commands:

```
Map<String, ShellCommand> commands = …;
commands.put(“exit”, new ExitShellCommand());
commands.put(“ls”, new LsShellCommand());
...
```

Then implement the shell as given by following pseudocode:

```
repeat {
  l = readLineOrLines
  String commandName = extract from l
  String[] arguments = extract from l
  command = commands.get(commandName)
  status = command.executeCommand(in, out, arguments)
} until status!=TERMINATE
```