CoffeeMaker

Application that simulates a coffee making machine.

Prerequisities

Before starting, you should have installed and properly configured the following:

- Git
- JDK 17+
- IntelliJ IDEA
- Internet connection

As you are working on university's workstation everything should be preconfigured for you.

General Requirements

- Console-based application
- Interactive (asks user to type input while running)
- Java 17+ SDK, no 3rd party libraries or databases required
- Tools like Ant, Maven, JUnit, TestNG, Guava are allowed, but not enforced
    - If you use them, remember about good and bad practices.
- Git repository with more than just initial and final commit, we wish to see your progress.
- Git repository contains a text file with an instruction how to run the solution.

> This exercise is NOT about using any external toolsets or frameworks. JDK classes and good, object-oriented design are enough to provide a successful solution.

Functional Requirements

Commit often, every 30 minutes. Don't think too deeply, delivering early steps matters, you have few hours for entire task.

Test how your application works, behaves and if output is aligned with the specification. It can be done via launching your application via IDE, providing commands from the example screens in the specification and checking if output is as expected.

> Remember that after each step the previous ones should still work as before. You should not lose applications functionalities when you are developing a new ones.

Step 0: User interactions

On startup, application asks the user to specify a command to execute. User shall type the command on the keyboard and hit Enter to execute. After executing the command, the application shall await another command.

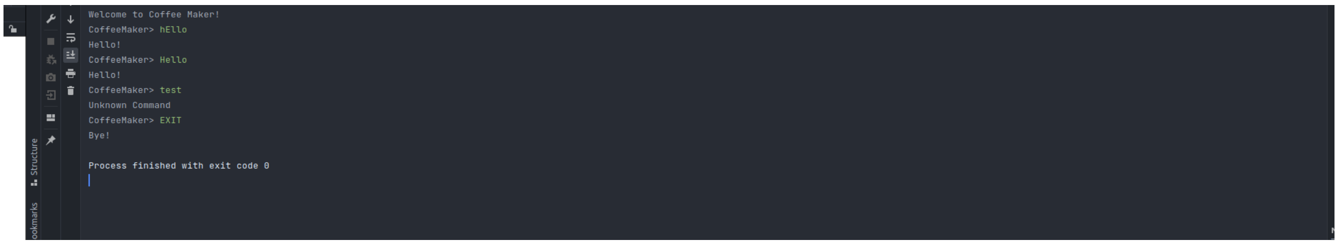
Example:

```
CoffeeMaker> Hello [Enter] ①②
Hello!
CoffeeMaker> Exit [Enter]
Bye!
$③
```

① command to execute

② hit the ENTER to pass the typed command to the application

③ that’s a system prompt - it means CoffeeMaker program ended



```
Welcome to Coffee Maker!
CoffeeMaker> hEllo
Hello!
CoffeeMaker> Hello
Hello!
CoffeeMaker> test
Unknown Command
CoffeeMaker> EXIT
Bye!

Process finished with exit code 0
```

Figure 1. An example application’s output from IntelliJ IDEA

Command names are case-insensitive. If the user provides a command that is unrecognized by the application, a response displayed by the application should be: “Unknown command!”, and the application should wait for another command.

Using imperative coding (if - else - if… in main) you could complete it within 10 minutes. It would be sufficient to deliver the step, but not in a very good manner. Try to follow an object-oriented paradigm to make your application is open for future extension (next steps).

Step 1: Status

When the user issues the `Status` command, the program prints the current level of each ingredient stored in the coffee machine.

Example:

```
CoffeeMaker> Status
Coffee: 54% 1080g
Water:  78% 1170g
Milk:   15% 150g
Cocoa:  0%  0g
CoffeeMaker>
```

The initial status of all ingredients is generated randomly when the application starts. Max. capacity of the reservoirs:

- Coffee: 2000g
- Water: 1500g
- Milk: 1000g
- Cocoa: 2000g

Step 2: Making coffee

Making coffee consists of the following steps:

1. Grind coffee (11 grams, 10 secs)
2. Heat water (200 grams, 15 secs)
3. Pour water (30 grams, 20 secs)
4. Pour water (170 grams, 10 secs)

When the user issues a `Make Coffee` command, the program goes through the recipe (step by step):

1. Prints the name of the stage on the screen,
2. Consumes the given amount of the ingredient,
3. Waits the given amount of time.

Example:

```
CoffeeMaker> Make Coffee
Grind coffee... [waits 10 secs] ①
Heat water...   [waits 15 secs] ①
Pour water...   [waits 20 secs] ①
Pour water...   [waits 10 secs] ①
Done!
CoffeeMaker>
```

① an amount of time that application should wait is in `[]`; it shouldn't be printed

If there is not enough of some ingredient, the program reports an error with a meaningful message and cancels the process. Cancellation returns to normal prompt and awaits next command - it DOES NOT crash the program with a stack trace.

Remember that after a coffee was made, given quantity of a specific ingredient should be deducted from its current level.

Step 3: Loading a recipe

When the user issues a `Make <Beverage>` command, program reads the recipe from a text file named `<Beverage>.rcp`, located in the current working directory, and having the following structure:

```
<Stage1>:<Ingredient1>:<Amount1>:<Time1>
<Stage2>:<Ingredient2>:<Amount2>:<Time2>
<Stage3>:<Ingredient3>:<Amount3>:<Time3>
...
```

And then makes the beverage according to the recipe.

Sample recipe (stored in the file named: CoffeeWithMilk.rcp):

```
GRIND:COFFEE:11:10
HEAT:WATER:180:15
POUR:WATER:180:30
HEAT:MILK:20:10
POUR:MILK:20:5
```

Sample session:

```
CoffeeMaker> Make CoffeeWithMilk
Grind coffee... [waits 10 secs] ①
Heat water...   [waits 15 secs] ①
Pour water...   [waits 30 secs] ①
Heat milk...    [waits 10 secs] ①
Pour milk...    [waits 5 secs] ①
Done!
CoffeeMaker>
```

① an amount of time that application should wait is in `[]`; it shouldn’t be printed

If the file doesn’t exist or doesn’t have the expected structure, an error message: "Invalid recipe file!" is shown, and the program waits for the next instruction.

Step 4: Cancelling

Pressing `Q` or `q` on the keyboard while making a coffee causes the machine to stop brewing and wait for another command do execute.

Example:

```
CoffeeMaker> Make Coffee
Grind coffee... [waits 10 secs]
Heat water...   [waits 15 secs]
q
Cancelled!
CoffeeMaker>
```

It is possible to implement this feature without using threads. If you are unsure how to use multithreading, it is better not to use them.

Other Requirements

- Changes shall be pushed frequently to the remote GIT repository (every 30 min).
- Each GIT commit should provide a meaningful message containing a short description of the changes.
- It is recommended that the code contained in each commit compiles and works.

Tips:

- Review the provided examples. They show quite precisely what your application should do.
- Take some time to think about the solution. Make good use of the paper and the pen.
- Questions are welcome.