Specification

🐢 Program Behavior

Our `Turtle` based domain specific language, which we have named *TurtleTalk*, specifies the basic commands that you need to interact with the Turtle environment in a meaningful way. A list of these commands and their descriptions can be found below:

- Forwards < X >
  - Moves the `Turtle` forwards by X amount, where < X > is some "chunk" of input data that should be read as a `double`
- Backwards < X >
  - Moves the `Turtle` backwards by X amount, where < X > is some "chunk" of input data that should be read as a `double`
- Left < X >
  - Turns the `Turtle` left by angle X, where < X > is some "chunk" of input data that should be read as a `double`
- Right < X >
  - Turns the `Turtle` right by angle X, where < X > is some "chunk" of input data that should be read as a `double`
- Up
  - Lifts the `Turtle` from the canvas, stopping it from drawing
- Down
  - Places the `Turtle` back onto the canvas, allowing it to draw again
- ChangeColor < Y >
  - Changes the color of the line produced by the `Turtle` to Y, where < Y > is the name of the color found in the collapsible list below. You should read the name in as a `String`

Expand

"aqua", "black", "blanchedalmond", "blue", "brown", "chocolate", "crimson", "cyan", "darkblue", "darkgray", "darkgreen", "darkred", "gold", "gray", "green", "indigo", ""lightblue", "lightgreen", "lightgray", "magenta", "navy", "orange", "pink", "purple", "red", "silver", "tomato", "violet", "white", "yellow"

An example *TurtleTalk* program can be found below, with its expected output in the `Turtle` environment after. (This code is also in the file `instructions.txt` in the coding slide.) Expand to see the full execution:

Expand

Input File:

Turtle Environment Output:

Your program should read from a file containing *TurtleTalk* commands line by line, and execute each command as it reads it. When the end of the file is reached the program can simply stop execution.

> We encourage you to play with the commands in `instructions.txt` to see what other output you can create!

📂 Reading from a File

You should read commands from the file `instructions.txt`. You will find a partially complete program in the scaffold of the next slide. Notice that the `Scanner` over this file and the `Turtle` to execute the commands on have already been created for you at the top of the `main` method. You should use both of these constructs in your code, and should not create

any additional `Turtle`s. You also should not create any additional `Scanner`s *over the input file,* though you may create `Scanner`s for other purposes (such as to process a `String`).

Each line of any input file is guaranteed to contain a single command. Since a command might contain more than one token that is relevant for its execution, you should process each line as a whole using the `nextLine()` method of the file `Scanner`.

💻 Processing Individual Commands

The scaffold code also contains a `while` loop with some code inside of it. You should read in one line from the file per iteration of this while loop and process it fully within the bounds of this loop. After reading in a line, you should create a `Scanner` over it in order to process this line token by token, as shown in class. You should then read in the first token of this line using the `next()` method from said `Scanner`. The first token of each line will always be the text instruction for the command, and you should store it in a `String` variable named *command*.

1️⃣ Task 1 - Add Basic Commands

You should add to the conditional structure within the `while` loop to contain a case for each of the seven commands listed above in the Program Behavior portion of the spec. The `"Forwards"` command has already been implemented for you, and you can use this format to process the remaining commands.

🚨 Validation of Commands

You may **not** assume that all *TurtleTalk* commands you read are valid commands and properly formed. If a command in the instructions file does not exist as a command in your *TurtleTalk* language, you should print out to the console:

```
Error: invalid command
```

> Once you finish adding the basic commands, you can try Marking your code. Our test case checks that your basic commands can be used for the `instructions.txt` file. Make sure to read the "✅ **A Note About Submitting Your Work**" section at the top of the Coding Workspace's Description.

2️⃣ Task 2 - Expand the Turtle Interpreter!

For the second task of this assignment, you need to choose **two** of the following options to implement in your program.

🖌️ Option 1 - Draw Random-Colored Rectangle

For this option, you will implement a command called `DrawRectangle` that will draw a rectangle with a given width, height, **and optional color parameter**. When the optional color parameter is given in the file, draw the rectangle with the given color. When the color parameter is not given, you should randomly select a color for your rectangle.

Click Expand to see an example:

Expand

For the random color selection, you should decide on a palette of colors from which you will select one randomly. You may choose whatever collection of colors you want but there must be **at least 3** different possible colors (e.g. red, green, and blue) to randomly choose from. How you choose to select a random color is ultimately up to you, but one possible method is to map each color to a number (e.g. red=0, green=1, blue=2) and randomly generate 0, 1, or 2 to select red, green, or blue for the pen color.

You are **required to write a method** that takes in your Turtle, the rectangle's width, and the rectangle's height as parameters and draws the rectangle. This method should be called in the while loop in `main` accordingly, such that when the `DrawRectangle` command is encountered in the instructions file, the Turtle draws the rectangle. Don't forget to change the pen color (to either the specified color or a randomly selected color, depending on the command) before calling your method which draws the rectangle.

If you want to test that your code for this option works, you can try adding a `DrawRectangle` command to `instructions.txt` and click Run to see what gets produced.

🗣️ Option 2 - Draw Regular Polygon Using Console Scanner

For this option, you will implement a new command called `PromptUser`, which will prompt the user for a series of inputs in the console to draw a regular polygon. You should prompt the user for the number of sides, side length, color, and the number of shapes to draw.

Click Expand to see an example:

Expand

Below is a sample interaction in the console, after the `PromptUser` command was found in the instructions file. (Note that user input is **bolded and underlined**.)

```
How many sides? 5 What is the side length? 50 What color? green How many shapes? 3
```

Given these series of inputs, your Turtle could draw

You can draw the shapes in any configuration you would like.

The angle that the Turtle should turn after each side can be found using the following formula:

```
double angle = 360.0 / numSides
```

For example, using this formula, the angle that the Turtle would need to turn each time to make a pentagon would be:

```
360.0 / 5 = 72.0
```

You are encouraged to add more Scanner prompts to be more creative but you must have **at least** the 4 prompts listed above.

You are also **required to write a method** that takes in `Scanner console` and your Turtle as parameters. It should prompt the user for the series of inputs and then draw the regular polygons using those inputs. This method should be called in the while loop in `main` accordingly, such that when the `PromptUser` command is encountered in the instructions file, the Turtle draws the regular polygons.

> If you want to test that your code for this option works, you can try adding a `PromptUser` command to `instructions.txt` and click Run to see what gets produced.

✍️ Option 3 - Handling Comments

For this option, you will keep track of the number of comments seen in the instructions file and print out the number of lines of comments found throughout the file at the end to the console. Your code should handle **regular (single-line) and muti-line comments.**

For example:

Suppose we have the following instructions.txt file

Given this file, you should print out

```
Number of comments found: 8
```

after processing the entire instructions file. Notice that for multi-line comments, all of the code inside the `/* */` should be read as comments from your interpreter. This means your

interpreter should not execute `Forward 100` and `ChangeColor Yellow` from the example above. You may assume the opening `/*` and closing `*/` will **always appear in a new line by themselves** and there will always be **at least one space** after the `//` for single-line comments.

> If you want to test that your code for this option works, you can add regular comments and multi-line comments to `instructions.txt` and click Run to double check that (1) all comments in the instructions file do not get interpreted as commands, and (2) the correct number of lines of comments is printed out at the end.

🎢 Option 4 - Choose your own adventure!

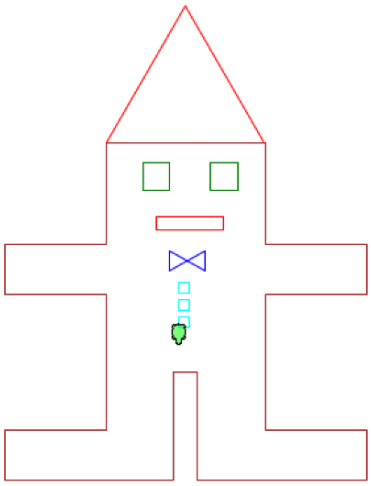
This must involve innovation beyond repeating something you were already asked to do in the concrete exercise. If you want to get credit for this option, describe your idea on this discussion board thread to get approval no later than **11:59pm on Sunday, February 26**. Approval means that your idea, *if executed well*, can earn credit for this option -- it does not guarantee any particular grade.

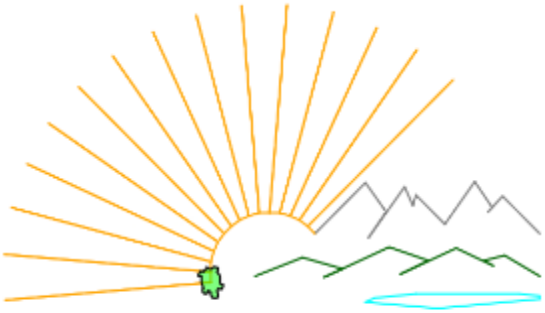
3️⃣ Application - Create a Picture From a File

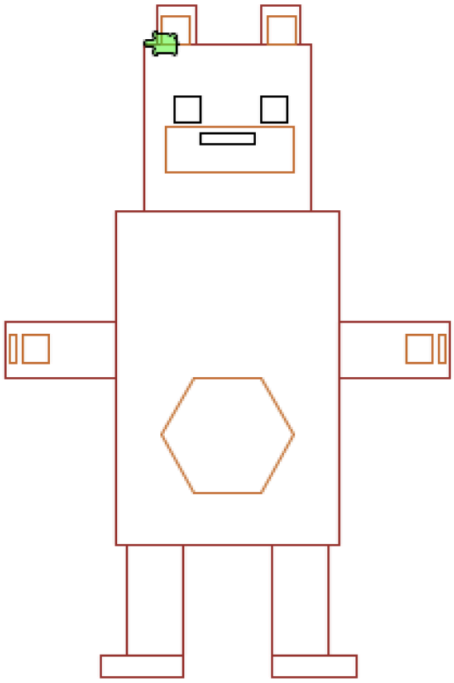
Use *TurtleTalk* and your Interpreter to create a picture! You should write and submit your *TurtleTalk* code in the `my_picture_instructions.txt` file found in the file browser of the next slide.

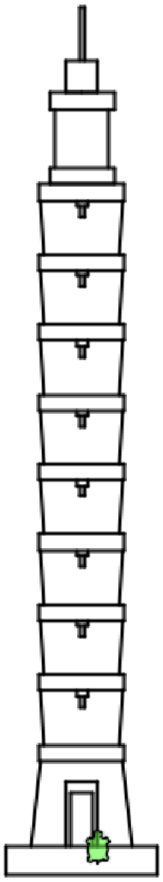
Click Expand to see example pictures:

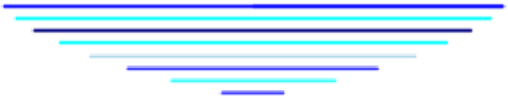
Expand

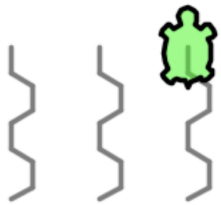


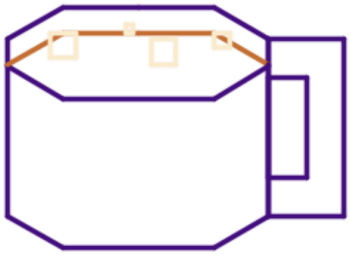


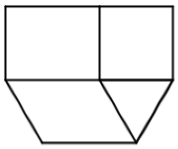












You are **required to use your two new commands from Task 2** to help draw your picture. We also highly encourage you to create additional commands to help draw your picture and make your life easier. For example, you could create a command to draw a square with some parameter, or a command to move the Turtle to a different location.

> A "picture" can just about be anything - you can choose to draw an object, a shape, etc. Although we encourage you to produce cool drawings, we are not assessing your artistic talent or your design abilities🎨! In fact, we encourage you to put your main focus of attention to the **quality of your code** rather than the quality of your drawings.

When you are ready to run your `my_picture_instructions.txt` instructions, make sure to modify the file name on line 8 of the scaffold from `instructions.txt` to `my_picture_instructions.txt`:

```
8 Scanner fileScan = new Scanner(new File("my_picture_instructions.txt"));
```

Development Strategy

As on previous assessments, you will likely want to approach the program one part at a time, rather than trying to write most or all of the program at once. We recommend the following approach:

- Begin by filling in the bound on the `while` loop in the scaffold, remembering that each iteration of the `while` loop should process a **single line** of the file.
- Then, read a line from the file, create a `Scanner` over it and read in the instruction portion of the command.
- Once you have that working, finish the `"Backwards"` command and implement the 5 remaining basic commands. You should also take care of error-checking invalid commands. You should extend the given conditional structure to do this. This completes Task 1.
- Next, extend the TurtleTalk language by choosing two of the four options in Task 2.
    - Remember that for Options 1 and 2 you need to create a method.
    - Consider creating your own txt file to test your Task 2 code!
- Finally, produce a picture for Task 3 using your TurtleTalk language commands. Make sure to use your two new commands from Task 2 to help draw your picture.

You should write the instructions for producing your picture in the provided `"my_picture_instructions.txt"` file.