INTRO

- Building a minimalistic shell for UNIX-like OS's

WHAT IS A SHELL

- Basic UI to your OS
- You can input commands to the shell and receive corresponding output

THE INPUT LOOP

- Keyboard is our standard input device (os.Stdin) and we can create a reader to access it

```
reader := bufio.NewReader(os.Stdin)
```

- Each time we press the enter key, a new line is created
- While pressing the enter key, everything written is stored in the variable input

```
input, err := reader.ReadString('\n')
```

- Let's put this in a main function
- By adding a for loop around the ReadString function, we can input commands continuously
- When an error occurs while reading the input, we will print it to the standard error device (os.Stderr)

```
func main() {
    reader := bufio.NewReader(os.Stdin)
    for {
        // Read keyboard input
        input, err := reader.ReadString('\n')
        if err != nil {
            fmt.Fprintln(os.Stderr, err)
        }
    }
}
```

EXECUTING COMMANDS

- We want to execute the entered command
- A new function called execInput which takes a string as an argument
- First, we have to retrieve the newline control character \n at the end of the input

- Then we prepare the command with exec.Command(input) and assign the corresponding output and error device for this command
- Lastly, the prepared command is processed with cmd.Run()

```
func execInput(input string) error {
    // Remove newline character
    input = strings.TrimSuffix(input, "\n")

    // Prepare command to execute
    cmd := exec.Command(input)

    // Set correct output device
    cmd.Stderr = os.Stderr
    cmd.Stdout = os.Stdout

    // Execute the command and return the error
    return cmd.Run()
}
```

FIRST PROTOTYPE

- Complete our main function by adding an input indicator (>>) at the top of the loop
- Also add the new execInput function at the bottom of the loop

```
func main() {
    reader := bufio.NewReader(os.Stdin)
    for {
        fmt.Print(">> ")
        // Read keyboard input
        input, err := reader.ReadString('\n')
        if err != nil {
            fmt.Fprintln(os.Stderr, err)
        }

        // Handle the execution of the input
        if err = execInput(input); err != nil {
            fmt.Fprintln(os.Stderr, err)
        }
    }
}
```

- Build and run the shell with go run main.go

ARGUMENTS

- Currently, we don't distinguish between the command and the arguments
- We have to modify the execLine function and split the input on each space

```
func execInput(input string) error {
    // Remove newline character
    input = strings.TrimSuffix(input, "\n")

    // Split the input to separate the command and the arguments
    args := strings.Split(input, " ")

    // Pass the program and the arguments separately
    cmd := exec.Command(args[0], args[1:]...)
```

- The program is now stored in args[0] and the arguments in the subsequent indices
- We should now be able to run commands like ls -l

CHANGE DIRECTORY (CD)

- Now we can run commands with an arbitrary number of arguments
- To have a set of functionality which is necessary for a minimal usability, we should implement changing the directory
- We have to modify the execInput function
- Just after the Split function, we add a switch statement on the first argument, which is stored in args[0]
- When the command is cd, we check if there are subsequent arguments, otherwise, we cannot change to a not given directory
- When there is a subsequent argument in args[1], we change the directory with os.Chdir(args[1])
- At the end of the case block, we return the execInput function to stop further processing of this built-in command
- Because of its simplicity, we can just add a built-in exit command right below the cd block, which stops our shell