Solution Review: Map Polar Points to Cartesian Points

This lesson discusses the solution to the challenge given in the previous lesson.

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package main
import (
        "bufio"
        "fmt"
        "math"
        "os"
        "runtime"
        "strconv"
        "strings"
)

type polar struct {
        radius, θ  float64  // greek character!
}

type cartesian struct {
        x, y float64
}

const result = "Polar: radius=%.02f angle=%.02f degrees -- Cartesian: x=%.02f y=%.02f\n"

var prompt = "Enter a radius and an angle (in degrees), e.g., 12.5 90, " + "or %s to quit."

func init() {
        if runtime.GOOS == "windows" {
                prompt = fmt.Sprintf(prompt, "Ctrl+Z, Enter")
        } else { // Unix-like
                prompt = fmt.Sprintf(prompt, "Ctrl+C")
        }
}

func main() {
        questions := make(chan polar)
        defer close(questions)
        answers := createSolver(questions)
        defer close(answers)
        interact(questions, answers)
}
```

```
}

func createSolver(questions chan polar) chan cartesian {
        answers := make(chan cartesian)
        go func() {
                for {
                        polarCoord := <-questions
                        0 := polarCoord.0 * math.Pi / 180.0 // degrees to radians
                        x := polarCoord.radius * math.Cos(0)
                        y := polarCoord.radius * math.Sin(0)
                        answers <- cartesian{x, y}
                }
        }()
        return answers
}

func interact(questions chan polar, answers chan cartesian) {
        reader := bufio.NewReader(os.Stdin)
        fmt.Println(prompt)
        for {
                fmt.Printf("Radius and angle: ")
                line, err := reader.ReadString('\n')
                if err != nil {
                        break
                }
                line = line[:len(line)-1] // chop off newline character
                if numbers := strings.Fields(line); len(numbers) == 2 {
                        polars, err := floatsToStrings(numbers)
                        if err != nil {
                                fmt.Fprintln(os.Stderr, "invalid number")
                                continue
                        }
                        questions <- polar{polars[0], polars[1]}
                        coord := <-answers
                        fmt.Printf(result, polars[0], polars[1], coord.x, coord.y)
                } else {
                        fmt.Fprintln(os.Stderr, "invalid input")
                }
        }
        fmt.Println()
}

func floatsToStrings(numbers []string) ([]float64, error) {
        var floats []float64
        for _, number := range numbers {
                if x, err := strconv.ParseFloat(number, 64); err != nil {
                        return nil, err
                } else {
                        floats = append(floats, x)
                }
        }
        return floats, nil
}
```

Click the **RUN** button, and wait for the terminal to start. Type `go run main.go` and press **ENTER**.

The above program includes *two* major *structs*:

- `polar`: takes *two* `float64` variables as fields: `radius` and `θ`.

- `cartesian`: takes *two* `float64` variables as fields: `x` and `y`.

Line 20 defines a *constant* `result` string for the formatted output, and **line 22** defines a string `prompt` for the command-line. This program also shows the use of an init function (from **line 24** to **line 30**). This is used here to test for the operating system on which the program runs, and changes the prompt variable accordingly (that's why it had to be a var).

In `main()`, **line 33** makes a channel `questions` of type `polar`, and closes this at the end of the program with a `defer` at **line 34**. At **line 35**, `questions` is then passed as a parameter to function `createSolver()`, which returns a channel of type `cartesian`, which is captured in variable `answers` at **line 35**. `answers` is also closed at the end of the program with a `defer` at **line 36**.

`Function createSolver()` makes a *local* channel `answers` at **line 41**. Then, it starts an anonymous function call (from **line 42** to **line 50**). In an infinite for-loop starting at **line 43**, a `polarCoord` struct is taken from the channel `questions`. Using the given mathematical functions, the polar coordinates are converted to cartesian coordinates from **line 45** to **line 47**. Then, the `cartesian` struct is made with the transformed coordinates and put on the `answers` channel. Then, at **line 51**, the channel of the cartesian values is returned.

Finally, at **line 37**, the `interact()` function is called, passing the *two* channels `questions` and `answers` as parameters. At **line 55**, the program makes a buffered reader to read from the keyboard. In an infinite for-loop (from **line 57** to **line 63**), it reads a string with polar coordinates (**line 59**) with error-handling (from **line 60** to **line 76**). It splits up the string on spaces to a string array `numbers`. As long as there are 2 fields (`len(numbers)==2`), it converts the strings to *polars* at **line 65** with `floatsToStrings()`. Look at the header of this function at **line 80**. It converts each string into a float and returns an array of floats, or a possible error while converting.

Then, back at **line 70**, we put a struct of polar coordinates onto the `questions` channel. At **line 71**, we read the cartesian coordinates from the `answers` channel and print them together at **line 72**. Note that **invalid number** and **invalid input** messages are printed to the error output at **line 67** and **line 74**,

respectively, without leaving the for-loop.

That is it about the solution. In the next lesson, we'll discuss how to handle communication over time.