Go Flow Control

Loops, If/Else, Switch, and Defer

Part 2: The Go Series

1. The 'For' Loop

Go has only **one** looping construct: the `for` loop. It uses three components separated by semicolons: init, condition, and post.

```go
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
fmt.Println(sum)
```

2. 'For' is Go's 'While'

You can drop the init and post statements. At that point, `for` behaves exactly like a `while` loop in other languages.

```go
sum := 1
for sum < 1000 {
    sum += sum
}
fmt.Println(sum)
```

3. Infinite Loops

If you omit the loop condition, it loops forever. This is commonly used for servers or listening for events.

```go
func main() {
    for {
        fmt.Println("Looping forever...")
        // break // needed to stop
    }
}
```

4. If Statements

Go's `if` statements are like C or Java, but the parentheses `( )` are gone and the braces `{ }` are required.

```go
func sqrt(x float64) string {
    if x < 0 {
        return sqrt(-x) + "i"
    }
    return fmt.Sprint(math.Sqrt(x))
}
```

5. If with Initialization

A powerful feature: `if` can start with a short statement. Variables declared here are **only available inside the if scope**.

```go
if v := math.Pow(x, n); v < lim {
    return v
} else {
    fmt.Printf("%g >= %g", v, lim)
}
```

6. Switch Statements

A cleaner way to write sequence if-else. **Crucial:** Go only runs the selected case, not all the following ones (no `break` needed).

```
switch os := runtime.GOOS; os {
case "darwin":
    fmt.Println("OS X.")
case "linux":
    fmt.Println("Linux.")
default:
    fmt.Printf("%s.", os)
}
```

7. Switch with No Condition

Switch without a condition is the same as `switch true`. This is the cleanest way to write long if-then-else chains.

```go
t := time.Now()
switch {
case t.Hour() < 12:
    fmt.Println("Good morning!")
case t.Hour() < 17:
    fmt.Println("Good afternoon.")
default:
    fmt.Println("Good evening.")
}
```

8. Defer Statement

A `defer` statement delays the execution of a function until the surrounding function returns. Great for cleanup.

```go
func main() {
    defer fmt.Println("world")

    fmt.Println("hello")
}
// Output: hello world
```

9. Stacking Defers

Deferred calls are pushed onto a stack (LIFO). When a function returns, its deferred calls are executed in **reverse order**.

```go
fmt.Println("counting")

for i := 0; i < 4; i++ {
    defer fmt.Println(i)
}

fmt.Println("done")
// Output: counting, done, 3, 2, 1, 0
```