Conditionals & Loops

02-201 / 02-601
Conditionals
If Statement

if statements let you execute statements conditionally.

```go
func max(a int, b int) int {
    var m int
    if a > b {
        m = a
        fmt.Println(a)
    } else {
        m = b
        fmt.Println(b)
    }
    return m
}
```

condition

“then” part: executed if the condition is TRUE

“else” part: executed if the condition is FALSE

else part is optional.

If statements let you make choices:
if the condition is true, the else part will be skipped;
if the condition is false, the then part will be skipped.
### Conditions

**Boolean Operator**

<table>
<thead>
<tr>
<th>Boolean Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>e</code></td>
<td><code>e_1</code></td>
</tr>
<tr>
<td><code>e</code></td>
<td><code>e_1</code></td>
</tr>
<tr>
<td><code>e</code></td>
<td><code>e_1</code></td>
</tr>
<tr>
<td><code>e</code></td>
<td><code>e_1</code></td>
</tr>
<tr>
<td><code>e</code></td>
<td><code>e_1</code></td>
</tr>
<tr>
<td><code>e</code></td>
<td><code>e_1</code></td>
</tr>
<tr>
<td><code>!e</code></td>
<td>true if and only if <code>e</code></td>
</tr>
</tbody>
</table>

**Example conditions:**

```go
if a > b {
    m = a
    fmt.Println(a)
} else {
    m = b
    fmt.Println(b)
}
```

`e_1` and `e_2` can be complicated expressions

**Boolean expressions:** because they evaluate to true or false

- `a > 10 * b + c`
- `10 == 10`
- `square(10) < 101 - 1 + 2`
- `!(x*y < 33)`
Boolean Operators: AND and OR

<table>
<thead>
<tr>
<th>Boolean Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>true if e</td>
</tr>
<tr>
<td>e</td>
<td>true if e</td>
</tr>
</tbody>
</table>

“pipe” character | Often above \ on your keyboard

Examples, true or false?

\[a=10\\ b=50\]

- \(a > 10 \land b > 20\) \(\text{false}\)
- \(a=10 \land b < 100 \land a \cdot b > 1000\) \(\text{false}\)
- \(a>20 \lor b < 51 \lor b-a \cdot b > 0\) \(\text{true}\)
- \(a=10 \land b=50\) \(\text{syntax error!}\)
- \(a=10 \land b >= 100 \lor b == 50\) \(\text{true}\)

Boolean Expressions:

- \(b==50 \lor a == 10 \land b >= 100\) \(\text{true}\)
- \(a>5 \land b>20 \lor a==0 \land b==0\) \(\text{true}\)
- \(a>5 \land b>20 \land a==0 \land b==0\) \(\text{true}\)
- \(a>5 \land (b>20 \land a==0) \land b==0\) \(\text{true}\)
Example “if” statements

```go
// max() returns the larger of 2 ints
func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

// max() returns the larger of 2 ints
// equivalent to above
func max(a, b int) int {
    if a > b {
        return a
    } else {
        return b
    }
}

if temperature > 100 {
    fmt.Println("Warning: too hot!")
}

var a, b int = 3, 3

if a < 10 {
    a = a * a
}

if a * a > 3 * b {
    t := a
    a = b
    b = t
}

if a < b {
    fmt.Println(a)
} else {
    fmt.Println(b)
}

// AbsInt() computes the absolute value of an integer.
func AbsInt(x int) int {
    if x < 0 {
        return -x
    }
    return x
}
```

Q: What will this print?
A: 3
Another If Example

// returns the smallest even number
// among 2 ints; returns 0 if both are odd
func smallestEven(a, b int) int {
    if a % 2 == 0 {
        if b % 2 == 0 {
            // both a and b are even, so
            // return smaller one
            if a < b {
                return a
            } else {
                return b
            }
        } else {
            // only a is even
            return a
        }
    } else if b % 2 == 0 {
        // only b is even
        return b
    } else {
        // both a and b are odd
        return 0
    }
}
Switch statement

`switch` statements let you express several, mutually exclusive tests compactly.

```go
// even() returns the smallest even number // among 2 ints; returns 0 if both are odd
func smallestEven(a, b int) int {
    switch {
    case a % 2 && b % 2:
        if a < b {
            return a
        } else {
            return b
        }
    case a % 2 == 0:
        fmt.Println("Returning a")
        return a
    case b % 2 == 0:
        return b
    default:
        return 0
    }
}
```

Each `case` part contains a condition, followed by a “:” and then a sequence of statements.

The statements associated with the *first true* case will be executed.

Q: would it be ok to swap the first and second cases in smallestEven()?  

No!

The optional `default` case is executed if none of the others are.

`switch` statements in Go are much more powerful than those in Java, C, and C++.
Why are they called switch statements?

**Analogy**: a railroad switch: depending on the condition of the switch, the train will go down a different track.

![Railroad Switch Diagram](image)
**General Switch Statements**

The first case that contains an expression that equals the switch expression will execute.

```go
switch a*a {
    case 2,4,6,8,10:
        fmt.Println("Square of a is even!")
    case 1,3,5,7,9,b*b:
        fmt.Println("Square of a is odd or equals b squared!")
    default:
        fmt.Println("Variable a is <= 0 or > 10")
}
```

expressions in cases need not be constants
Example

Convert a character of DNA into an integer representation:

```go
// acgt() takes a letter and returns the index in 0,1,2,3 to which it is
// mapped. 'N's become 'A's and any other letter induces a panic.
func acgt(a byte) byte {
    switch a {
    case 'A':
        return 0
    case 'N':
        return 0
    case 'C':
        return 1
    case 'G':
        return 2
    case 'T':
        return 3
    }
    panic(fmt.Errorf("Bad character: %s!", string(a)))
}

We'll see byte, string, and panic later.
Loops
Loops

- Loops let you repeat statements.
- The statements in the body of the loop will be executed until the loop condition is false.
- Go has only “one” kind of loop: the `for` loop, with 2 different forms.

Initialization statement: executed once *before* the loop starts

The condition: the loop continues until this is false.

```
func factorial(n int) int {
    var f int = 1
    for i := 1; i <= n; i = i + 1 {
        f = f * i
    }
    return f
}
```

Each time through the loop is an *iteration*

Post-iteration statement: executed *after each time through the loop*.
“while” loops

- You can omit the initialization statement and the post-iteration statement in a `for` loop.

- This form is sometimes called a “while” loop, because it loops “while the condition is true”

- These two code snippets are *almost* equivalent:

```go
var f int = 1
for i := 1; i <= n; i=i+1 {
    f = f * i
}
```

```go
var f int = 1
i := 1
for i <= n {
    f = f * i
    i = i + 1
}
```

- Can you guess the difference?

**Answer: Scope! of the `i` variable**

In the first: the `i` variable’s scope is only the body of the `for` loop
In the second: `i` lasts until the end of the enclosing scope
For Loop Control Flow

```go
var f int = 1
for i := 1; i <= n; i=i+1 { 
    f = f * i
}
```
Variable Definitions in Loop Bodies

What will the following function print?
Is it correct?

```go
func sumSquares() {
    // print partial sums of the sequence of squares
    // of the numbers 1 to 10
    for i := 1; i <= 10; i = i + 1 {
        var j int
        j = j + i * i
        fmt.Println(j)
    }
}
```

This is wrong!
It will print:

1
4
9
16
25
36
49
64
81
100

which are the first 10 squares, not their sums

Why?

Variable `j` is created and destroyed each time through the loop
Nested loops: Printing a “Square”

```
func printSquare(n int) {
    for i := 1; i <= n; i=i+1 {
        for j := 1; j <= n; j=j+1 {
            fmt.Print("#")
        }
        fmt.Println(""
    }
}
```
Example: Random Walks

Simulate a random walk on an n-by-n chessboard
**Example: Random Walks**

Simulate a random walk on an n-by-n chessboard

```go
func randDelta() int {
    return (rand.Int() % 3) - 1
}

func randomWalk(n, steps int) {
    var x, y = n/2, n/2
    fmt.Println(x, y)
    for i := 0; i < steps; i++ {
        var dx, dy int
        for dx == 0 && dy == 0 {
            dx = randDelta()
            for x+dx < 0 || x+dx >= n {
                dx = randDelta()
            }
        }
        dy = randDelta()
        for y+dy < 0 || y+dy >= n {
            dy = randDelta()
        }
        x += dx
        y += dy
        fmt.Println(x, y)
    }
}
```

`rand.Int()` returns a random non-negative integer.

Must put

```go
import "math/rand"
```

at top of your program.

Loop to make sure we move.

Loop to keep position within [0, n) x [0, n)

Note the code duplicating the test for an in-field coordinate.

This isn't very good.

Better to break this out into a function.
This version is:

- clearer
- more flexible — perhaps we can use randStep() someplace else.
- Slightly shorter (25 vs. 26 lines)
Example: Print a Diamond

`func printDiamond(n, shift int)`

- `shift` = number of characters to shift diamond right
- `n` = number of lines (must be odd)

Break into two subproblems:
- `printTriangle(n, shift int)`
- `printInvertedTriangle(n, shift int)`

`printDiamond(19, 5)`
Example: printDiamond

Break into two subproblems:

printTriangle(n, shift int)
printInvertedTriangle(n, shift int)

```go
func printDiamond(n, shift int) {
    if n % 2 == 0 {
        fmt.Println("Error! n must be odd; it's", n)
    } else {
        printTriangle(n / 2 + 1, shift)
        printInvertedTriangle(n/2, shift+1)
    }
}
```

Check that the parameters are valid. This is good practice.

Print top triangle.

Print bottom triangle.

Since \( n \) is odd:

\[
\left\lfloor \frac{n}{2} \right\rfloor = \frac{n}{2} + 1
\]

What's going on here?

Since \( n \) is an integer variable and 2 is an integer the code \( n \div 2 \) does integer division and rounds down.

The bottom triangle is slightly shorter and shifted to the right by 1 extra space.
Top-Down Program Design

- We “used” the `printTriangle()` and `printInvertedTriangle()` functions in our thinking before we wrote them.

- We know what they are supposed to do, so we could use them to write `printDiamond()` even before we implemented them.

- In a sense, it doesn’t matter how `printTriangle()` and `printInvertedTriangle()` are implemented: if they do what they are supposed to do, everything will work.

- It’s only their **interface** to the rest of the program that matters.

- This is top-down design, and it’s often a very good way to approach writing programs:
  1. start by breaking down your task into subproblems.
  2. write a solution to the top-most subproblem using functions for other subproblems that you will write later.
  3. then repeat by writing solutions to those subproblems, possibly breaking *them* up into subproblems.
Good Programming:

Break big problems into small functions with good interfaces.
The size variable tracks the number of # to print on the current row.

size goes up by 2 after each row

The loops for n rows (0 to n-1) are:

```
for row := 0; row < n; row = row + 1 {
    // print space to indent row
    for i := 1; i <= (n - 1) - row + shift; i = i + 1 {
        fmt.Print(" ")
    }
    // print the right number of symbols in a row
    for i := 1; i <= size; i = i + 1 {
        fmt.Print("#")
    }
    size = size + 2
    fmt.Println()
}
```

Print a newline (return) character after each row

Tip: watch out for “off-by-one” errors: e.g. using row <= n or row := 1 (though using both would be ok)
Why n - row - 1 + shift?

for i := 1; i <= (n - 1) - row + shift; i = i + 1 {
    fmt.Print(" ")
}

when row = n-3, loop should execute 2 + shift times
when row = n-2, loop should execute 1 + shift times
when row = n-1, loop should execute shift times

At each row, one fewer space should be written. The last row (numbered n-1) should have shift spaces written.
func printInvertedTriangle(n, shift int) {
    var size int = 2*n - 1
    // Note: this loop counts down
    for row := n; row > 0; row = row - 1 {
        for i := 1; i <= n - row + shift; i = i + 1 {
            fmt.Print(" ")
        } // print the right number of symbols in a row
        for i := 1; i <= size; i = i + 1 {
            fmt.Print("#")
        }
        size = size - 2
        fmt.Println()
    }
}
func printTriangle(n, shift int) {
    var size int = 1
    for row := 0; row < n; row = row + 1 {
        // print space to indent row
        for i := 1; i <= n - row - 1 + shift; i = i + 1 {
            fmt.Print(" ")
        }
        // print the right number of symbols in a row
        for i := 1; i <= size; i = i + 1 {
            fmt.Print("#")
        }
        size = size + 2
        fmt.Println()
    }
}

func printInvertedTriangle(n, shift int) {
    var size int = 2*n - 1
    // Note: this loop counts down
    for row := n; row > 0; row = row - 1 {
        for i := 1; i <= n - row + shift; i = i + 1 {
            fmt.Print(" ")
        }
        // print the right number of symbols in a row
        for i := 1; i <= size; i = i + 1 {
            fmt.Print("#")
        }
        size = size - 2
        fmt.Println()
    }
}

func printDiamond(n, shift int) {
    if n % 2 == 0 {
        fmt.Println("Error! n must be odd; it's", n)
    } else {
        printTriangle(n / 2 + 1, shift)
        printInvertedTriangle(n/2, shift+1)
    }
}
A worse way to write `printDiamond()`

```go
func badPrintDiamond(n, shift int) {
    if n % 2 == 0 {
        fmt.Println("Error! n must be odd; it's", n)
    } else {
        var size int = 1
        for row := 0; row < n/2+1; row = row + 1 {
            // print space to indent row
            for i := 1; i <= (n/2+1) - row - 1 + shift; i = i + 1 {
                fmt.Print(" ")
            }
            // print the right number of symbols in a row
            for i := 1; i <= size; i = i + 1 {
                fmt.Print("#")
            }
            size = size + 2
            fmt.Println()
        }
        size = n - 1
        for row := (n/2); row > 0; row = row - 1 {
            for i := 1; i <= (n/2) - row + shift+1; i = i + 1 {
                fmt.Print(" ")
            }
            // print the right number of symbols in a row
            for i := 1; i <= size; i = i + 1 {
                fmt.Print("#")
            }
            size = size - 2
            fmt.Println()
        }
    }
}
```

**Bug!** In fact, there is a subtle bug here:

Must understand the entire function before you really know what it does.
Bugs in top part affect execution of bottom part (what if you reassigned `n` accidentally someplace?)
Summary

- Conditionals let you choose which code to execute based on Boolean expressions.

- Go has two types of conditionals: `if...else` and `switch`.

- Loops execute a set of statements repeatedly while a Boolean expression is true and stop when it becomes false.

- Go has only one type of loop: `for`.

- Along with functions and variables, these constructs form the basis of all programs.