Carl Kingsford, 02-201, Fall 2015

Lecture 7: Conditionals and Loops

"If" Statements

`if` syntax

`if` statements let you execute statements conditionally: that is which statements execute depend on whether some condition is true or false. For example:

```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
}
```

The syntax of an `if` statement is:

```go
if CONDITION {
    THEN PART
} else {
    ELSE PART
}
```

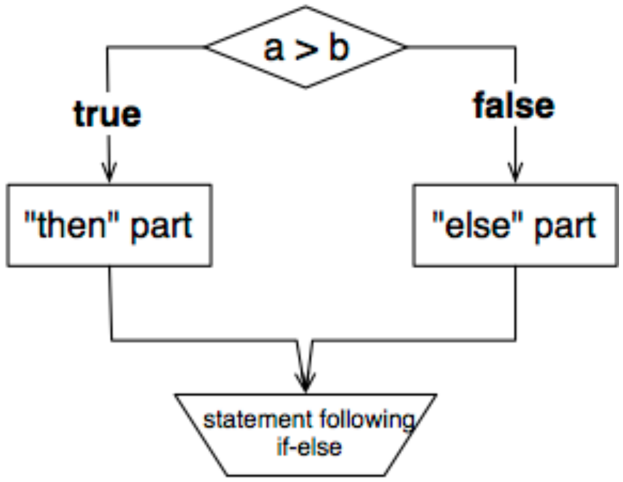
If `CONDITION` is true, then the statements in the `THEN PART` will be executed.

If `CONDITION` is false, then the statements in the `ELSE PART` will be executed.

The else clause is optional:

```go
if a > b {
   fmt.Println("a is bigger")
}
fmt.Println("done")
```

Schematically:



> Test yourself! What kind of computer instruction do you think plays an important role in how `if` statements are executed?

Conditions

The CONDITION part of an `if` statement can be any expression that evaluates to a Boolean value. For example, the following comparison operators return Boolean values.

Boolean Operator	Meaning
e1 > e2	e1 is greater than e2
e1 < e2	e1 is less than e2
e1 >= e2	e1 is greater than or equal to e2
e1 <= e2	e1 is less than or equal to e2
e1 == e2	e1 is equal to e2
e1 != e2	e1 is not equal to e2
!e1	true if and only if e1 is false

Examples:

```
a > 10 * b + c
10 == 10
square(10) < 101 - 1 + 2
!(x*y < 33)
```

The Boolean operators `&&` and `||` (**and** and **or**) are particularly useful in `if` conditions.

Additional `if` examples

A max function:

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

The same function can be written with an `else` clause too:

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

The else clause is optional, and you can have as many statements as you want inside the `THEN PART` and `ELSE PART`:

```
if temperature > 100 {
   fmt.Println(“Warning: too hot!”)
   fmt.Println("Run away!")
}
```

Go requires that

- the `{` must be on same line as the `if`
- `}` and `{` must be on same line as the `else`

Another example:

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

> **Test yourself!** What does the following print?
>
> ```go
> var a,b int = 3,3
>
> if a < 10 {
>   a = a*a
> }
> if a * a > 3*b {
>    var t int = a
>    a = b
>    b = t
> }
> if a < b {
>    fmt.Println(a)
> } else {
>    fmt.Println(b)
> }
> ```

`if` statements can be nested: an `if` statement can appear inside the THEN PART or ELSE PART of another `if` statement. This is common, and let's you make complex decisions.

```go
// 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
   }
}
```

Reminder: `%` is the "mod" operator: `x % y` is the remainder when `x` is divided by `y`.

Notice that you can put an `if` directly following an `else`: see line marked with a `***` above. This is the same as:

```go
if a % 2 == 0 {
    ...
} else {
   if b % 2 == 0 {
     ...
   }
}
```

but uses one fewer set of `{ }`.

"For" Loops

`for` syntax

Sometimes we want to execute a sequence of instructions many times. For this, a loop is what we need.

Go has only 1 kind of loop (with 2 variants): the `for` loop.

The statements in the body of the loop will be executed until the loop condition is false. Each time through the loop is called an iteration.

An example:

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

The syntax for a `for` loop is:

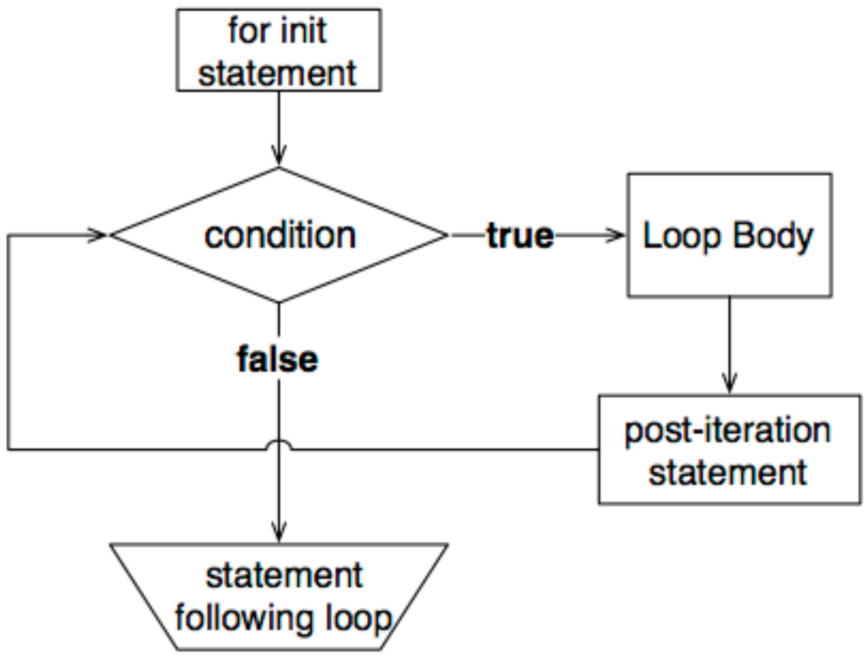
```
for INITIALIZATION_STATEMENT ; CONDITION ; POST-ITERATION_STATEMENT {

    FOR-BODY
}
```

There are 3 parts following the `for` and before the `{`, and these parts are separated by semicolons. These parts work as follows:

- INITIALIZATION_STATEMENT: a single statement that is executed one time before the loop starts.
- CONDITION: the loop will repeatedly execute until this condition is false.
- POST-INTERATION_STATEMENT: this is run after each time the FOR-BODY is executed

> **Test yourself!** How many times is the word "Hi" printed by this loop:
>
> ```go
> var i int
> for i = 10; i < 20; i = i+1 {
>   fmt.Println("Hi")
> }
> ```

Schematically, a `for` loop works like this:



Note: any of the three parts of the `for` statement can be omitted. If both the INITIALIZATION_STATEMENT and the POST-ITERATION_STATEMENT are omitted, you can omit the `;`

"While" loops

If we only include the CONDITION in a for loop, we will execute the FOR-BODY "while" CONDITION is true. You could re-write `factorial` as:

```
var f int = 1
var i int = 1
for i <= n {     // only condition in for
   f = f * i
   i = i + 1
}
```

Increment and Decrement operators

The `i++` and `i--` operators add 1 or subtract 1 from a variable. These are particularly useful in the POST-ITERATION_STATEMENT part of a `for` loop, where you can write `i++` instead of the longer (but exactly equivalent) `i = i + 1`.

:= variable declarations

Notice in the `factorial` example that we had to create a variable `i` that just served to count how many times we had executed the loop. This is quite common. Go provides a shorthand for this so that you can declare a variable inside of the INITIALIZATION_STATEMENT:

```go
v := 1
```

The `:=` operator both declares and initializes a variable. The above is equivalent to:

```go
var v int = 1
```

Question: In `v := 1`, how does Go know what type `v` is?

The answer is that Go knows `v` must be an integer because `1` is an integer. This works for `string`s and `float64`s too:

```go
r := 3.14159
s := "Hi there"
```

These statements save you typing `var` and the type. We can now rewrite `factorial` in a more clear, typical way:

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

> **Question:** What is the difference between these two snippets:
>
> ```go
> var f int = 1
> for i := 1; i <= n; i++ {
>    f = f * i
> }
> ```
>
> and

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

The answer is the *scope* of the variable `i`. In the first snippet, `i` lasts only for the loop, while in the second example, `i` lasts after the loop completes.

Variable declarations in loop bodies

What will the following function print? Is it correct?

```go
// BAD CODE
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)
    }
}
// BAD CODE
```

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 does this happen?

Variable `j` is created and destroyed each time through the loop!

Nested loops

Loops can be nested just like `if` statements. For example:

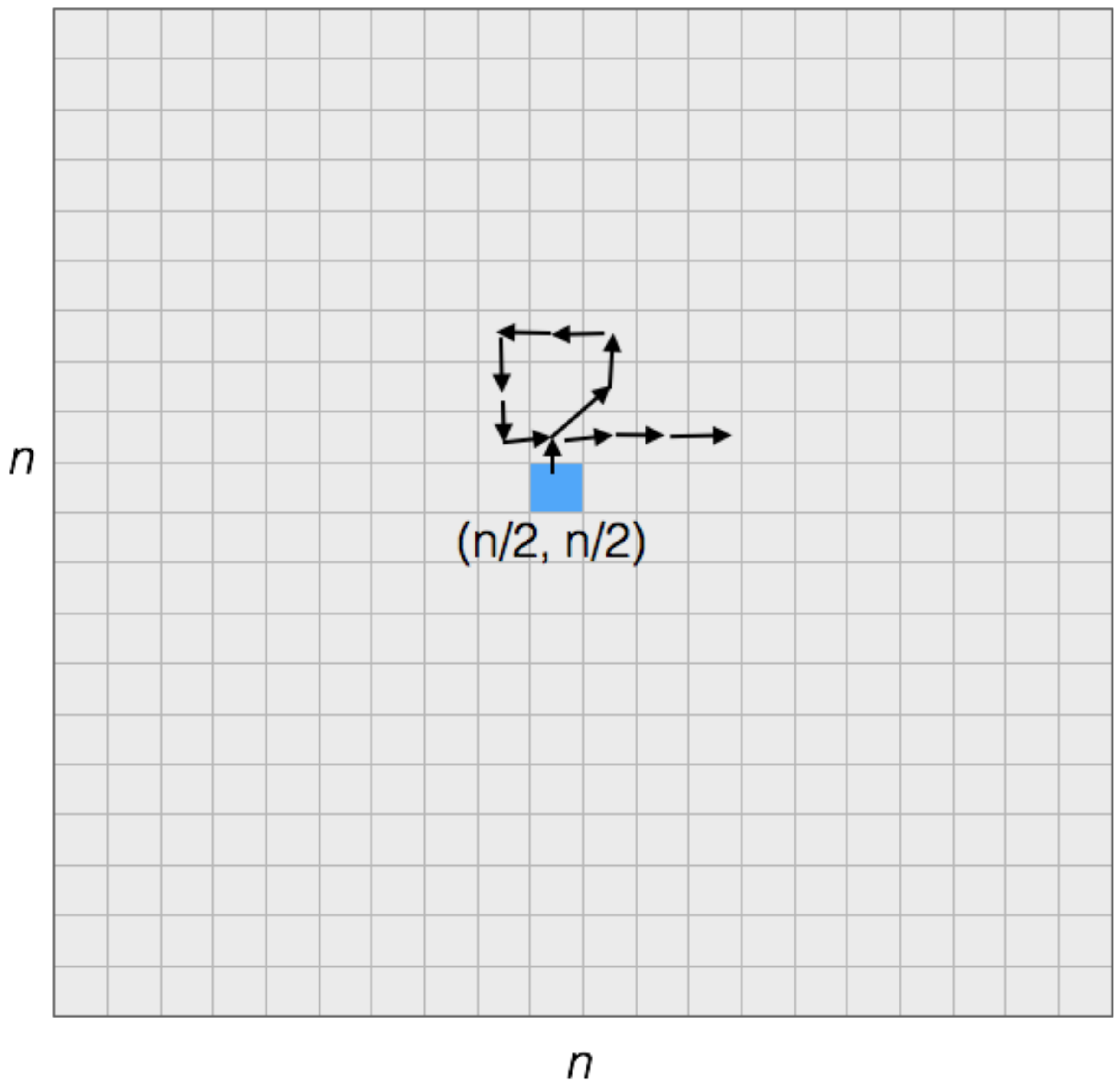
```go
func printSquare(n int) {
    for i := 1; i <= n; i=i+1 {
        for j := 1; j <= n; j=j+1 {
            fmt.Print("#")
        }
        fmt.Println("")
    }
}
```

will print:

```
carlk$ go run square.go
##########
##########
##########
##########
##########
##########
##########
##########
##########
##########
```

Example: Simulating Random Walks

A random walk on a grid is defined as follows: suppose you start standing at some square of the grid. You then choose an adjacent square to move to uniformly at random. You repeat this until you get tired. The sequence of squares you visit is called a random walk.



Let's write a function that will compute the squares you will visit if you perform a random walk on a n-by-n chess board, starting from the middle square.

Solution #1:

```go
package main
import (
    "fmt"
    "math/rand"
)

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

func randomWalk(n int, steps int) {
    var x, y = n/2, n/2
    fmt.Println(x,y)
    for i := 0; i < steps; i++ {           // A
        var dx, dy int

        for dx == 0 && dy == 0 {           // B
            dx = randDelta()
            for  x+dx < 0 || x+dx >= n {
                dx = randDelta()
            }

            dy = randDelta()
            for y+dy < 0 || y+dy >= n {    // C
                dy = randDelta()
            }
        }
        x = x + dx
        y = y + dy
        fmt.Println(x,y)
    }
}

func main() {
    randomWalk(10, 20)
}
```

Notes:

- `rand.Int()` returns a random non-negative integer. You must put `import "math/rand"` at top of your program to access that function.
- Loop A executes for the requested number of steps of the random walk

- Loop B executes until we successfully pick a non-zero direction to move in
- Loop C executes until we pick a y-direction that keeps us inside the chess board.

This works fine, and will print something like:

```
5 5
4 5
3 4
2 5
3 4
4 5
5 6
4 5
3 4
4 4
5 4
4 5
4 6
4 5
4 6
5 7
6 8
5 8
5 7
6 6
```

Solution #2

Note that the code around loop C is basically the same as the code just above it! This is code duplication and is usually bad. The reason it is bad is that if you fix a bug in one copy of the code, you have to remember to fix it in the other copy. In addition, it indicates you haven't broken your problem down into the best set of functions.

If we create a function for the idea of taking a random step, the code becomes shorter, easier to read, with little duplication:

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

func inField(coord, n int) bool {
    return coord >= 0 && coord < n
}

func randStep(x,y,n int) (nx int, ny int) {
    nx, ny = x, y
    for (nx == x && ny == y) || !inField(nx,n) || !inField(ny,n) {
       nx = x+randDelta()
       ny = y+randDelta()
    }
    return
}

func randomWalk(n, steps int) {
    var x, y = n/2, n/2
    fmt.Println(x,y)
    for i := 0; i < steps; i++ {
        x,y = randStep(x,y,n)
        fmt.Println(x,y)
    }
}
```

Now, `inField` expresses the test for whether a coordinate is in the chessboard, `randStep` provides one random step, and `randWalk` is just a loop asking for the next random step.

This is more understandable.

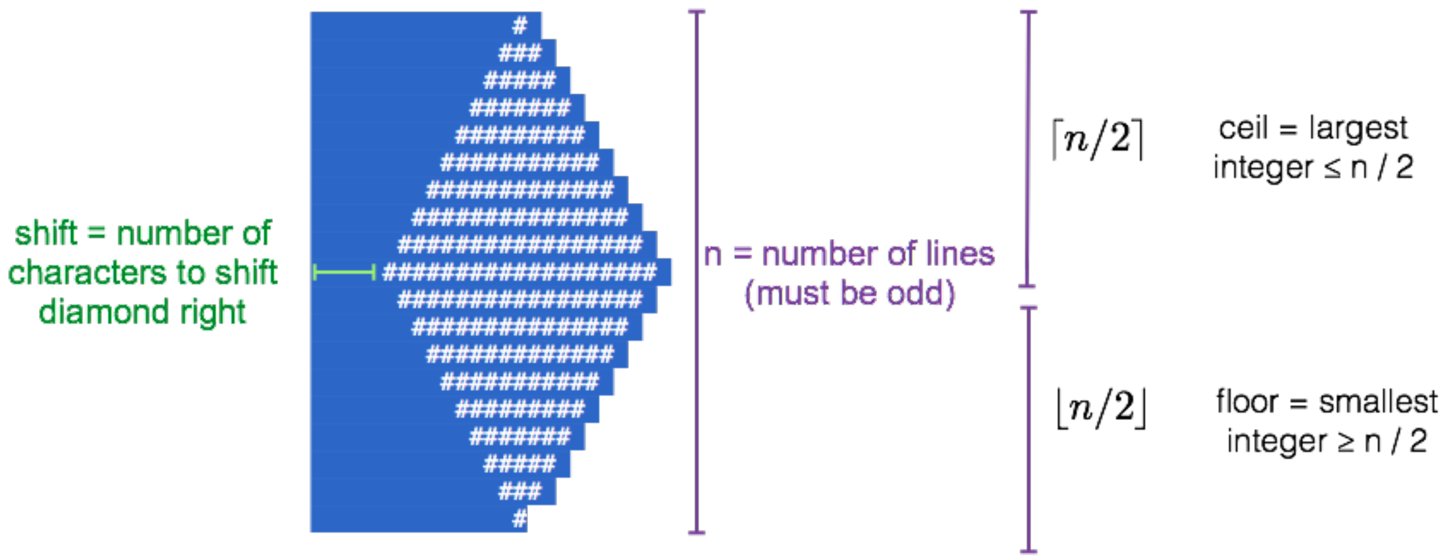
It is also more flexible: perhaps we will be able to use `randStep()` someplace else.

Good Programming: Break big problems into small functions with good interfaces.

Example: Printing a Diamond

Write a function to print out a diamond shape:

```
func printDiamond(n, shift int)
```

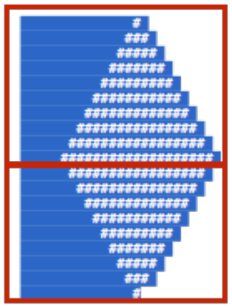

shift = number of characters to shift diamond right
n = number of lines (must be odd)
$\lceil n/2 \rceil$
ceil = largest integer ≤ n / 2
$\lfloor n/2 \rfloor$
floor = smallest integer ≥ n / 2

```
printDiamond(19,5)
```

How can we start thinking about this problem? Notice that a diamond is composed of two triangles. So we can break the problem down into two subproblems:

Break into two subproblems:

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



```
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:

$$\lceil n/2 \rceil = n/2 + 1$$
$$\lfloor n/2 \rfloor = n/2$$

What's going on here?
Since n is an integer variable and 2 is an integer
the code `n / 2` does ***integer*** division and rounds down.

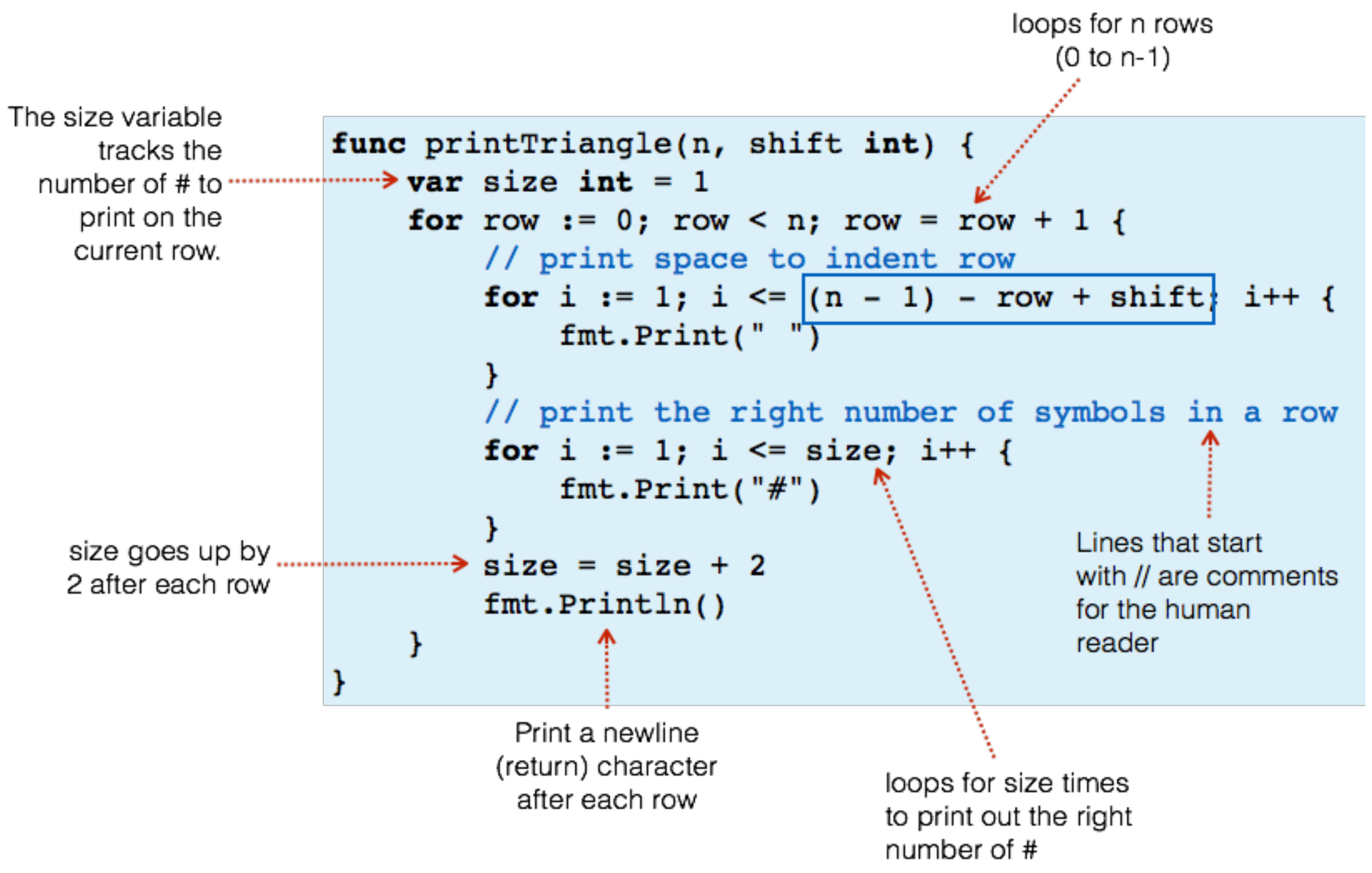
The bottom triangle is slightly shorter and shifted to the right by 1 extra space.

Top-down 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:
    - start by breaking down your task into subproblems.
    - write a solution to the top-most subproblem using functions for other subproblems that you will write later.

- then repeat by writing solutions to those subproblems, possibly breaking them up into subproblems.

The `printTriangle` function:

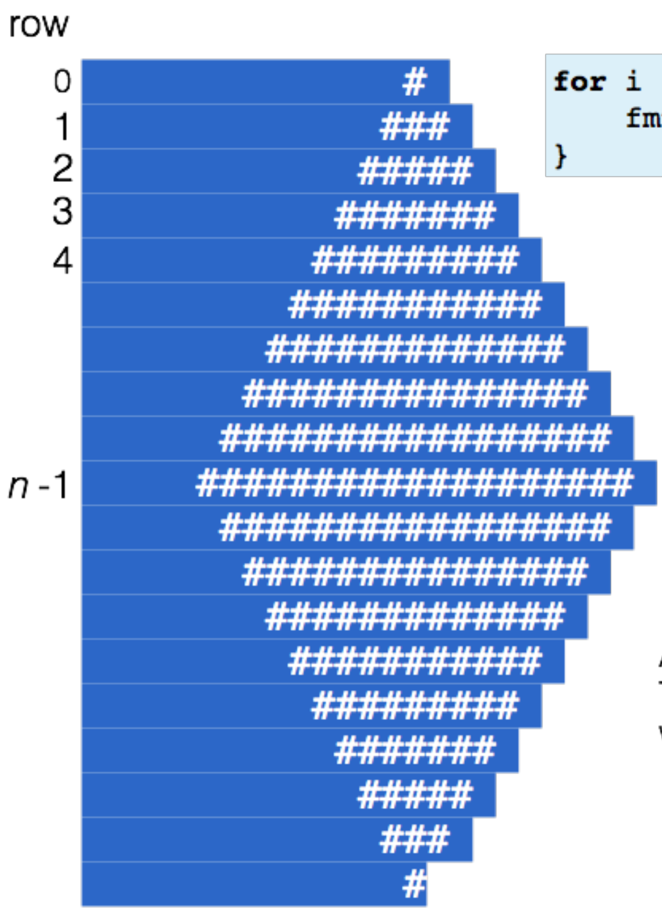


```go
func printTriangle(n, shift int) {
    var size int = 1
    for row := 0; row < n; row = row + 1 {
        // print space to indent row
        for i := 1; i <= (n - 1) - row + shift; i++ {
            fmt.Print(" ")
        }
        // print the right number of symbols in a row
        for i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

loops for n rows (0 to n-1)
The size variable tracks the number of # to print on the current row.
size goes up by 2 after each row
Lines that start with // are comments for the human reader
Print a newline (return) character after each row
loops for size times to print out the right number of #

Tip: watch out for "off-by-one" errors: e.g. using `row <= n` or `row := 1` (though using both would be ok)

Why `(n-1) - row + shift` ?



```
for i := 1; i <= (n - 1) - row + shift; i++ {
    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.

The `printInvertedTriangle` function:

size starts at the size of the top-most row, which has 2n - 1 symbols in it.

In first iteration of the row loop, row == n, so n - row = 0, and this loop iterates shift times

```
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++ {
            fmt.Print(" ")
        }
        // print the right number of symbols in a row
        for i := 1; i <= size; i++ {
            fmt.Print("#")
        }

        size = size - 2
        fmt.Println()
    }
}
```

The complete code for `printDiamond`:

```go
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)
    }
}
```

Summary

- Conditionals let you choose which code to execute based on Boolean expressions
- Conditionals in Go are expressed using the `if ... else` construct.
- 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.

Glossary

- iteration: one time through a loop.
- nested: when one programming construct is "inside" the body of another.
- code duplciation: duplicate code that is usually better put into a new function.
- top-down design: solving a big problem by breaking it into successively smaller problems.