1. The difference between 2 sides of the semicolon war was that while some programs use semicolon to have the mere effect of separating statements (for example, this allows us to write multiple statements on the same line like in Python: print(‘a’); print(‘b’); print(‘c’)), other programs see semicolon as an indicator of statement termination (basically indicates where an instruction ends and where the next instruction begins).

For usage of semicolons in Java and Go: in both programs, they are used as terminator, but unlike Java in which the semicolons needs to appear in the source, Go utilizes a tool to insert semicolons automatically as it scans, which is why the input code for Go rarely contains semicolons. That tool’s basic metaphor is to add a semicolon whenever a new line starts after a token/value that could terminate a statement. This also explains why in Java it is easy to add semicolon when writing a statement and go to the next line, while in Go mostly every statement has to be kept within a single line.

Example:

Java:

```
public static void main(String args[]) {

int ex1 = 11;
String ex2 = “ is the test number.”; System.out.println(ex1 + ex2);

}
```

Go

```
func() main {

ex1 := 11
ex2 := " is the test number."

fmt.Printf("%d %s\n", ex1, ex2) }
```

2. Thrown out. Non-compilation was due to a misunderstanding on my part. My apologies.

3. Go being statically typed means that the type is decided at writing time, not compile time. In languages like Java, this means explicitly writing out what that type is when you declare a

variable, however in Go this is not necessary because you implicitly declare a variable type when you assign it to a value.

Go can do this because among the common value types (ints, floats, Strings, bools, etc.), it is easy to see even with the naked eye the difference between them. For example, floats will always have a decimal, ints will be numbers without decimals, Strings have to have quotation marks, and bools are only true or false. The Go compiler is built to interpret these different hints and assign a variable its type (if not explicitly given) this way. Though this makes it appear dynamically typed because it physically writes similar to Python in that way, Python is dynamic because the variable types can change throughout the code, whereas they are static in Go, and therefore cannot change.

Personally, I am a fan of this method in Go, but mostly just because I am lazy. I get very tired of looking up different ways to initialize variable types with each language I learn since I sometimes have trouble remembering specific syntax differences between them. Remembering just to follow a simple structure of "variable := value" is much easier for me, so I consider it a win.

4. Java passes reference to array into subroutine, so changes made inside the function change the one instance of the the array.
Go passes COPY into subroutine, some changes made to array in sub do not affect original
Arrays created inside subroutine in Java are on heap, in Go on Stack.
On return, in Java, returns a reference. In Go, return is a copy of the thing from the stack (else it would go away). That is proof of return by value by analysis. In the example below, the function returns a closure and any array (yuck) in Go, because Go returns a copy, the array in the closure is independent of the array returned

```
package main

import "fmt"

func bbb() (func(), [3]int) {
    var arr [3]int
    arr[0]=5
    fmt.Printf("BBB %d\n", arr[0])
    clo := func() {
        fmt.Printf("clo %v\n", arr)
    }
    return clo, arr
}
func main() {
    clo, arr := bbb()
    clo()
```

```
    fmt.Printf("Main: %v\n", arr)
    arr[0] = 17
    fmt.Printf("Main: %v\n", arr)
    clo()

}
```

5.

GO: a: compile time pi compile time Main: ht: heap z stack m2: mm stack q heap r stack	Java p1 compile p2 compile vv pointer at compile, array from heap PP: ii stack Main: args heap (or stack) aa stack inst heap p3: prm stack all heap hmss heap

6. Methods are defined on types. Hence, the following is an equals method for the type TT

```
package main

import "fmt"

type TT []int

func (t TT) equals(slice2 TT) bool {
    if len(t) != len(slice2) {
        return false
    }
    for i := 0; i < len(t); i++ {
        if t[i] != slice2[i] {
            return false
        }
    }
return true
}
```

```
func main() {
   var a1,a2 TT
   for i:=0; i<10; i++ {
      a1 = append(a1, i)
      a2 = append(a2, i)
   }
   fmt.Printf("%t\n", a1.equals(a2))
}
```

7

- stack frame b(0)
  bval=9,
  resume to end of line 13 in func a
- Stack frame a(7)
  val = 7,
  resume to end of line 19 in func b
- Stack frame b(5)
  bval = 5,
  resume to end of line 13 in func a
- Stack frame a(3)
  val = 3,
  resume to end of line 19 in func b
- Stack frame b(1)
  bval = 1,
  resume to end of line 11 in func a
- Stack frame a(0)
  val = 0,
  resume to end of line 6 in main

- Stack Frame main()
  No local variables
  No resume point as no caller

Question 8:
1. Scope 1: Outside of all functions (1 to 24)

- Active: ss (global variable)
2. Scope 2: inside main function (5 to 13)

- Active:
+ ss (global variable) - only active from line 6 to line 7 and from line 10 to 12 + aa, ss (local variable inside of if statement) - only active from line 8 to line 9

3. Scope 3: inside first if statement's body block (line 7 to 10) - Active: aa, ss (local variable inside if statement)

4. Scope 4: in the first if statement (line 7) - Active: aa, ss (global variable)

5. Scope 5: inside first else statement (line 10 to 12) - Active: aa, ss (global variable)

6. Scope 6: inside f2 function (line 14 to 24)

- Active:
+ ss (local variable, declared in function f2) + f, qq (declared in the if statement)
+ qq (declared in the for loop)

7. Scope 7: inside the nested function (line 15 to 20)

- Active:
+ ss (local variable, declared in function f2) + qq (declared in the if statement)

8. Scope 8: inside the second if statement (line 16)
- Active: ss (local variable, declared in function f2)

9. Scope 9: inside the second if statement's body block (line 16 to 19)

- Active:
+ ss (local variable, initialized in function f2) + qq (declared in the if statement)

10. Scope 10: inside the for loop's body (line 21 to 23)

- Active:
+ ss (local variable, initialized in function f2) + qq (declared in the for statement)

11. Scope 11: in the for statement (line 21)

- Active:
+ ss (local variable, initialized in function f2) + qq (declared in the for statement)