Learn you a Go

Go

- Imperative/procedural: loops, assignment, pointers
- Statically + dynamically typed
- Garbage collected -- no explicit memory allocation/deallocation
- Program structuring (instead of OO): **structs + interfaces**
- Concurrency: lightweight threads with sync or async **message passing**
  → "goroutines" ←

Handling errors in Go: nil

nil $\in T$

- *nil* is the default "zero" value for data
- for pointers it's a *missing* pointer, causing a "panic" if used, with one exception:

```
data, err := ioutil.ReadFile(fileName)  // functions can return *two* values
if err != nil {                         // nil is good for error values
        fmt.Println("Error reading file:", err)
        return
}
```

Entity-Component-System (ECS) example

Video game entities (objects) have shared *components*, eg.

- `type Physics struct { Velocity, Mass, CollisionShape }`
  (field types omitted; capital letters for exported ids)
- `type Character struct { AggroAbility, Suspicion}`
- `type Health struct { Max, Current, Alive }`
- `type History struct { Encounters, Conversation }`
- `type Audio struct { Looping, DestructionEffect }`

How would you model this in Java?

- Each entity is some combo of these components e.g.
  - `type Rock *struct { Physics } // i.e. Rock includes Physics fields`
  - `type Monster *struct { Character, Health, Audio }`
  - `type NPC *struct { History, Audio }`
- OO class hierarchy: what component(s) would be right below `Entity`?

Conclusion: OO class hierarchy doesn't help here.

Idea (Go interfaces, Rust traits, Swift protocols, Haskell type classes):

- associate behaviour with a component
- automatically lift behaviour to entities having the component
- *duck typing*: if it methods like a duck and methods like a duck, it's a duck

~~Classes~~Struct+Interfaces

struct: like *class*, but just the data, no methods

interface: set of all types having the given methods

Property	Go interfaces	Java interfaces	Haskell type classes
declaration	implicit	explicit	explicit
typeckecking/resolution	static+dynamic	static	static
composable	yes	yes	yes
type *constructors*, multiparameter	no	no	yes

Interface/method basics

Basic interface declaration:

```
type T interface {
//   ^-- name of the interface
    m1(A1,...,An) B
//  |              ^-- type of method result
//  |  ^^^^^^^^^^-- types of method arguments
//  ^-- name of first method
    m2(...
    ⋮
    mk(...
}
```

Note the the "receiver"/"self"/"main" argument is not shown.

Method definition

T0=type to put in interface w. method m

```
func (p T0) m(x1,...,xn) T1 {
  ⋮
  return …
}
```

not in Java, instead use self instead of p

Basic types

built-in types: `int`, `float64`, `string`, `bool`, `byte`, ...

clones: e.g. in `type Bool bool`, the type `Bool` is distinct from `bool`

structs

in Haskell, Bool ≡ bool

Simple interface example (from Go documentation)

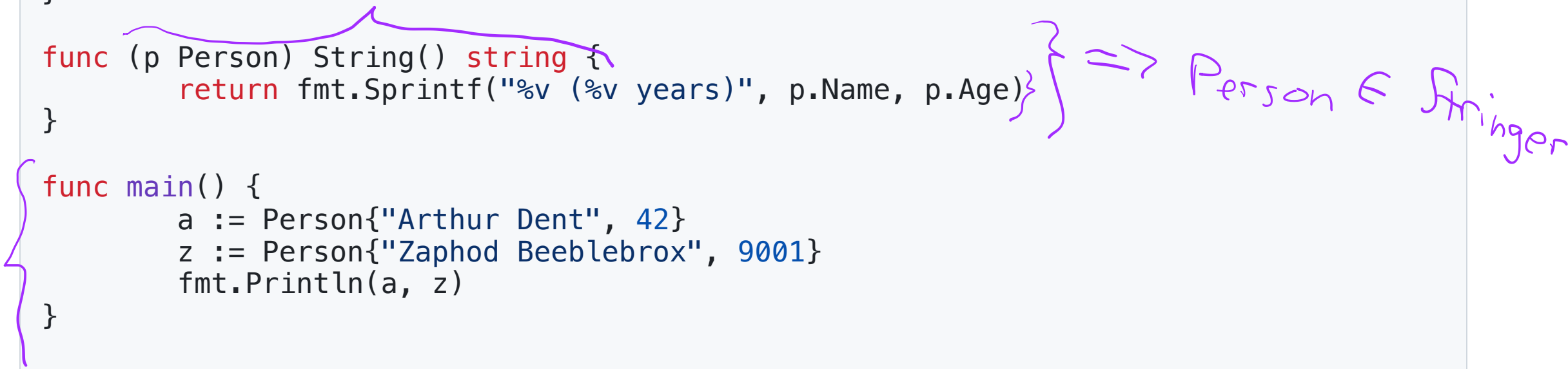
```
package main

type Stringer interface {
    String() string
}

type Person struct {
        Name string
        Age  int
}

func (p Person) String() string {
        return fmt.Sprintf("%v (%v years)", p.Name, p.Age)
}

func main() {
        a := Person{"Arthur Dent", 42}
        z := Person{"Zaphod Beeblebrox", 9001}
        fmt.Println(a, z)
}
```



Defining interfaces

```
type someInterface interface {
        <lines>
}
```

Each line defines a set of non-interface types (e.g. structs and base types).

The interface is the intersection of these sets.

Each line is one of the following.

1. A method → the set of all non-interface types with that method
2. A non-interface type → singleton set of that type
3. ~T for T a non-interface type → all non-interface types with underlying type T

~int

Examples

```
interface {
  int
}

interface {
  ~int
}

interface {
  ~int
  String() string
}

interface {
  int
  string
}
```