Review: example of structs + interfaces in Go.

```go
type Pointy interface {
        X() int
        Y() int
}

type Point struct {
        pointX, pointY int
}

func (p Point) X() int {
        return p.pointX
}

func (p Point) Y() int {
        return p.pointY
}

type PointWithZ struct {
        Point
        pointZ int
}
```

Review: interfaces

The meaning of an interface is a **set of concrete types** (i.e. non-interface types).

An interface definition specifies the set of concrete types that

1. have specified methods defined, *and*
2. are in some other specified interfaces, *and*
3. possibly, have one of a list of specified underlying types.

Pointers, r-values, l-values

Pointers are ubiqitous in Go programs.

We'll discuss them in the context of general imperative languages.

In general, imperative languages have two kinds of values.

- **l-values** are the values of the left-hand side of an assignment statement
- **r-values** are the values of the right-hand side of an assignment statement

A variable has both

- an l-value, which is the memory location it denotes, and
- an r-value, which is the value stored at that location

To run the assignment `x = y`:

- get the *memory location* (l-value) α denoted by x
- get the *memory location* β (l-value) denoted by y
- get the *value v* (r-value of y) stored at β
- store *v* at α

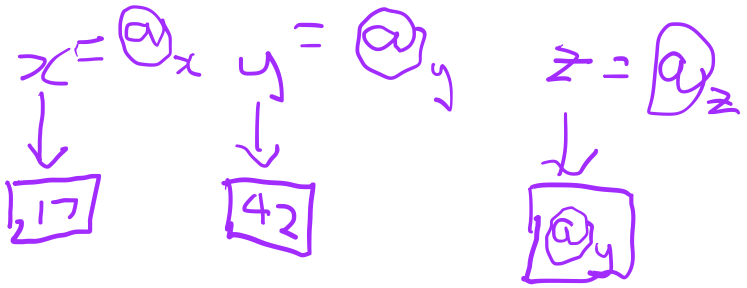
Exposing l-values

Some imperative languages, e.g. Go, expose l-values, i.e. memory locations, as data, usually calling them *pointers*.

> `*T` is the type of a addresses of, or pointers to, a value of type `T` in memory

The r-value of any expression is the result of evaluating it.
In addition, some expressions can have an l-value.

Expression	l-value	r-value
`*e`	(r-)value of `e`, requiring `e ∈ *T`	value stored at value of `e`
`&x`	none	address of x
`&e`	- (illegal expression if `e` not a variable)	-

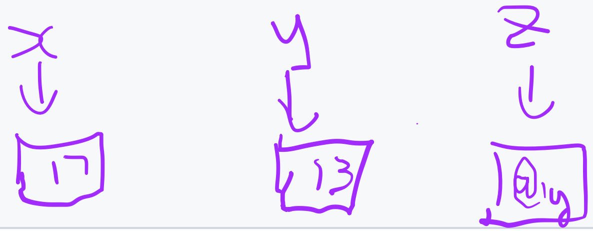


@ - addresses in memory

Example:

```
var x int = 17
var y int = 42
var z *int = &y
*z = 13
x = *z
```

13



Pointers and storage management: compare the two following data types.

```
type ok struct { x int; rest ok }
type notOk struct { x int; rest *ok }
```

could involve ton of copying

Some Go std-lib interfaces

```go
// package io
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}

type ReaderWriter interface {
        Reader
        Writer
}

// package sort
type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

Extended example: error

In Go's error package (std lib), errors are thought of as trees.

Leaf constructor: `errors.New : func(string) error`

Node constructor: `errors.Join : func(error, error) error`

Node accessor:

- `errors.Unwrap : func(error) error` or
- `errors.Unwrap : func(error) []error`

Query: `errors.Is : func(error, error) bool`

Get: `errors.As : func(error, interface{}) bool`

Error type is an interface

```go
type error interface {
        Error() string
}

func New(text string) error {
        return &errorString{text}
}

type errorString struct {
        s string
}

func (e *errorString) Error() string {
        return e.s
}
```