generic (or parametric) polymorphism

the ability for an entity to behave in the same way regardless of “input” or “contained” type
But what about (+)?

(+) :: Int -> Int -> Int

(+):= a -> a -> a
ad hoc polymorphism

the ability for an entity to behave differently on different “input” or “contained” types
typeclass

a set of types defined by an interface (set of functions) that the type must implement

or type constructors
class `Eq` a where
  (==) :: a -> a -> Bool

class `Show` a where
  show :: a -> String

> :type (==)
(==) :: Eq a => a -> a -> Bool

> :type show
show :: Show a => a -> String
Polymorphic values revisited

[] :: [a]

undefined :: a

1 :: Num a => a
Higher-order typeclasses
But what about `map`?

```
list-map :: [] a -> [] b
  (a -> b) -> [a] -> [b]

stream-map ::
  (a -> b) -> Stream a -> Stream b

vector-map ::
  (a -> b) -> Vector a -> Vector b
```
class Functor f where 
    fmap :: (a -> b) -> f a -> f b
Representing failing computations
NoMethodError in Rms::Properties#index

Showing /home/srcna/ruby/indieassets/app/views/rms/properties/index.html.erb where line #29 raised:

undefined method `name' for nil:NilClass

Extracted source (around line #29):

```html
    <td><%= property.seller_phone %></td>
    <td><%= property.category.name %></td>
    <td><%= property.property_name %></td>
    <td><%= property.property_details %></td>
    <td><%= property.location %></td>
```
C COMPILER

Y U NO TELL ME WHERE THE SEGMENTATION FAULT IS
In Haskell, types are “non-null.” If function f returns an Int, it can’t return “null” or “None”.

But sometimes we want to encode the possibility of failure.

data Maybe a = Nothing | Just a
Encoding failure: parsing integers

read :: Read a => String -> a

readMaybe :: Read a => String -> Maybe a
Chaining failing computations
s = null;
if (x != null) {
    y = x.f();
    if (y != null) {
        z = y.g();
        if (z != null) {
            for (a in z) {
                if (a != null) {
                    s = update(s, a);
                }}
        }
    }
}

if (s != null) {
    return s.h();
}
case x of
  Nothing  -> Nothing
  Just x'  ->
    case f x' of
      Nothing  -> Nothing
      Just y   ->
        case g y of
          Nothing  -> Nothing
          Just z   ->
            foldl ___ Nothing z
Some shortcuts from other languages

Elvis operator ?:  
  x ?: y

Safe navigation operator ?.  
  x?.y
Back to Haskell

couldFail :: _ -> Maybe _

Given $x = \text{couldFail} \ y$, what now?
“If null then null, else do something”

add10Maybe :: Maybe Int -> Maybe Int
add10Maybe Nothing = Nothing
add10Maybe (Just x) = Just (add10 x)

lengthMaybe :: Maybe [a] -> Maybe Int
lengthMaybe Nothing = Nothing
lengthMaybe (Just xs) = Just (length xs)
“If null then null, else do something”

add10Maybe :: Maybe Int -> Maybe Int
add10Maybe Nothing = Nothing
add10Maybe (Just x) = Just (add10 x)

lengthMaybe :: Maybe [a] -> Maybe Int
lengthMaybe Nothing = Nothing
lengthMaybe (Just xs) = Just (length xs)
try :: (a -> b) -> Maybe a -> Maybe b
try _ Nothing = Nothing
try f (Just x) = Just (f x)
try :: (a -> b) -> Maybe a -> Maybe b
try _ Nothing = Nothing
try f (Just x) = Just (f x)
Maybe is a functor!

\[
\begin{align*}
\text{try} & : (a \to b) \to \text{Maybe } a \to \text{Maybe } b \\
\text{fmap} & : (a \to b) \to f a \to f b
\end{align*}
\]
“If null then null, else do something that might fail”

recipMaybe :: Maybe Float -> Maybe Float
recipMaybe Nothing = Nothing
decipMaybe (Just x) = if x == 0
                        then Nothing
                        else Just (1 / x)

headMaybe :: Maybe [a] -> Maybe a
headMaybe Nothing = Nothing
headMaybe (Just xs) = if null xs
                       then Nothing
                       else Just (head xs)
“If null then null, else do something that might fail”

\[
\text{recipMaybe} :: \text{Maybe Float} \rightarrow \text{Maybe Float}
\]

\[
\text{recipMaybe Nothing} = \text{Nothing}
\]

\[
\text{recipMaybe (Just } x \text{)} = \begin{cases} 
\text{Nothing} & \text{if } x = 0 \\
\text{Just } \left(\frac{1}{x}\right) & \text{else}
\end{cases}
\]

\[
\text{headMaybe} :: \text{Maybe [a]} \rightarrow \text{Maybe a}
\]

\[
\text{headMaybe Nothing} = \text{Nothing}
\]

\[
\text{headMaybe (Just } xs \text{)} = \begin{cases} 
\text{Nothing} & \text{if } \text{null } xs \\
\text{Just } \left(\text{head } xs\right) & \text{else}
\end{cases}
\]
tryFail :: (a -> Maybe b) -> Maybe a -> Maybe b
tryFail _ Nothing = Nothing
tryFail f (Just x) = f x
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing `andThen` _ = Nothing
(Just x) `andThen` f = f x

(the backticks allow a binary function to be used infix)
x \ `andThen` f  
\ `andThen` g  
\ `andThen` h
Encoding error information with `Either`

data Either a b = Left a | Right b

We often use `Either String b` to represent a successful `(Right b)` value, or an error with message `(Left msg)`. 