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

typevar

typeclass constraint

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

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

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

```
> :type show
show :: Show a => a -> String
```

Polymorphic values revisited

```
[] :: [a]

undefined :: a

1 :: Num a => a
```

[a]

↓

[] a

Set a

Higher-order typeclasses

But what about `map`?

[] a → [] b

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

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

vector-map ::
  (a -> b) -> Vector a -> Vector b
```

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Representing failing computations

Acción ▾ Herramientas ▾ Ctrl+Alt+Supr

```
Segmentation fault
Segmentation fault
/bin/sh: error while loading shared libraries: ■ F8
cannot open shared object file: No such file or dir
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
```

Minecraft 1.8

Connection Lost

Internal Exception: java.lang.NullPointerException

Back to title screen

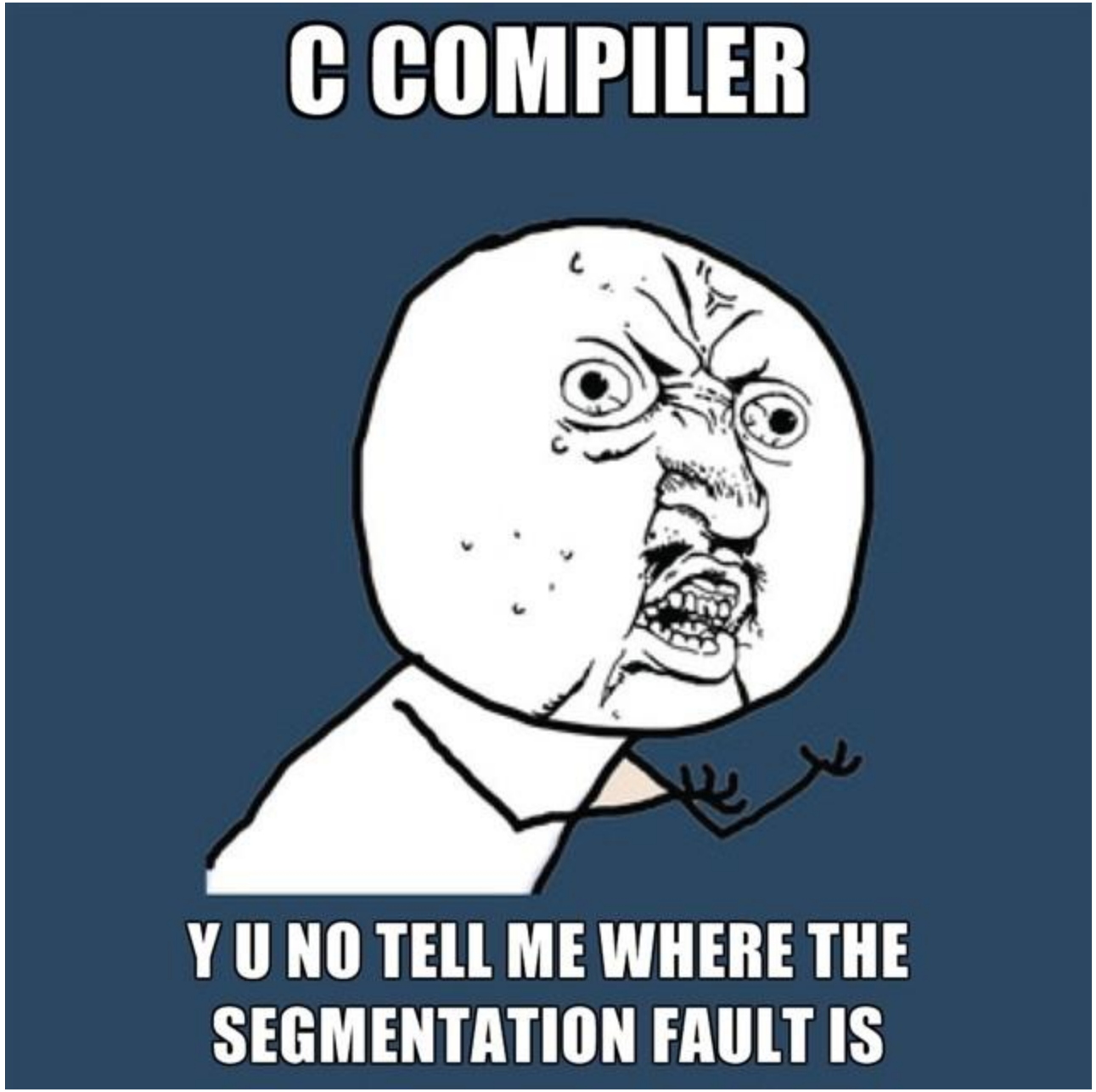
NoMethodError in Rms::Properties#index

Showing */home/sreena/ruby/indiaassets/app/views/rms/properties/index.html.erb* where line **#29** raised:

```
undefined method `name' for nil:NilClass
```

Extracted source (around line **#29**):

```
27        <td><%= property.seller_phone %></td>
28
29        <td><%= property.category.name %></td>
30        <td><%= property.property_name %></td>
31        <td><%= property.property_details %></td>
32        <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
```

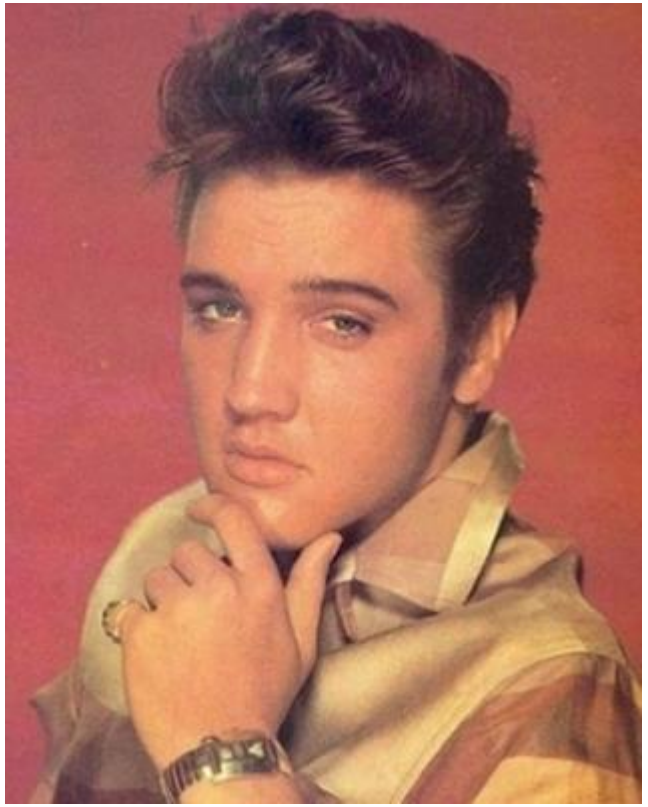
Some shortcuts from other languages

Elvis operator `?:`

```
x ?: y
```

Safe navigation operator `?.`

```
x?.y
```



Back to Haskell

```
couldFail :: _ -> Maybe _
```

Given `x = 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!

```
try ::  (a -> b) -> Maybe a -> Maybe b
fmap :: (a -> b) ->     f a ->     f b
```

“If null then null, else do something **that might fail**”

```
recipMaybe :: Maybe Float -> Maybe Float
recipMaybe Nothing = Nothing
recipMaybe (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**”

```
recipMaybe :: Maybe Float -> Maybe Float
recipMaybe Nothing = Nothing
recipMaybe (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)
```

```haskell
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)`.