Intro to Haskell

April 12, 2012

Speaker

Eric Normand

job: Scrive http://scrive.com

twitter: @ericnormand

email: ericwnormand@gmail.com

Slides

available at http://lispcast.com/haskell-slides

A language that doesn't affect the way you think about programming, is not worth knowing.

A language that doesn't affect the way you think about programming, is not worth knowing.

– Alan Perlis

why?

- get started in Haskell
- speak intelligently about Haskell
- get a feel for Haskell
- learn from my mistakes

especially if you typically use dynamic languages

me

- <u>not</u> an expert
- working > 1 year in Haskell
- learned on the job
- prefer dynamic languages
- appreciate static checking

Haskell

- static & strong type system
- purely functional
- lazy evaluation

Haskell

- static & strong type system
- purely functional
- lazy evaluation

- significant whitespace
- compiled
- garbage collected
- pattern matching
- from academia (but still practical)

type system

- static typing
  - types known at compile time
  - some explicit
  - some inferred
  - thrown away at compile time

type system

- static typing
  - types known at compile time
  - some explicit
  - some inferred
  - thrown away at compile time
- strong typing
  - all expressions have a type

type system

- static typing
  - types known at compile time
  - some explicit
  - some inferred
  - thrown away at compile time
- strong typing
  - all expressions have a type

if the compiler cannot determine the exact type, compilation fails

purely functional & lazy

- functions are values (with types)

purely functional & lazy

- functions are values (with types)
  - higher order

purely functional & lazy

- functions are values (with types)
  - higher order
- no mutable state by default

purely functional & lazy

- functions are values (with types)
  - higher order
- no mutable state by default
  - new values
  - names can be used once in a scope

purely functional & lazy

- functions are values (with types)
  - higher order
- no mutable state by default
  - new values
  - names can be used once in a scope
- lazy evaluation

purely functional & lazy

- functions are values (with types)
  - higher order
- no mutable state by default
  - new values
  - names can be used once in a scope
- lazy evaluation
  - “nothing” computed until needed

purely functional & lazy

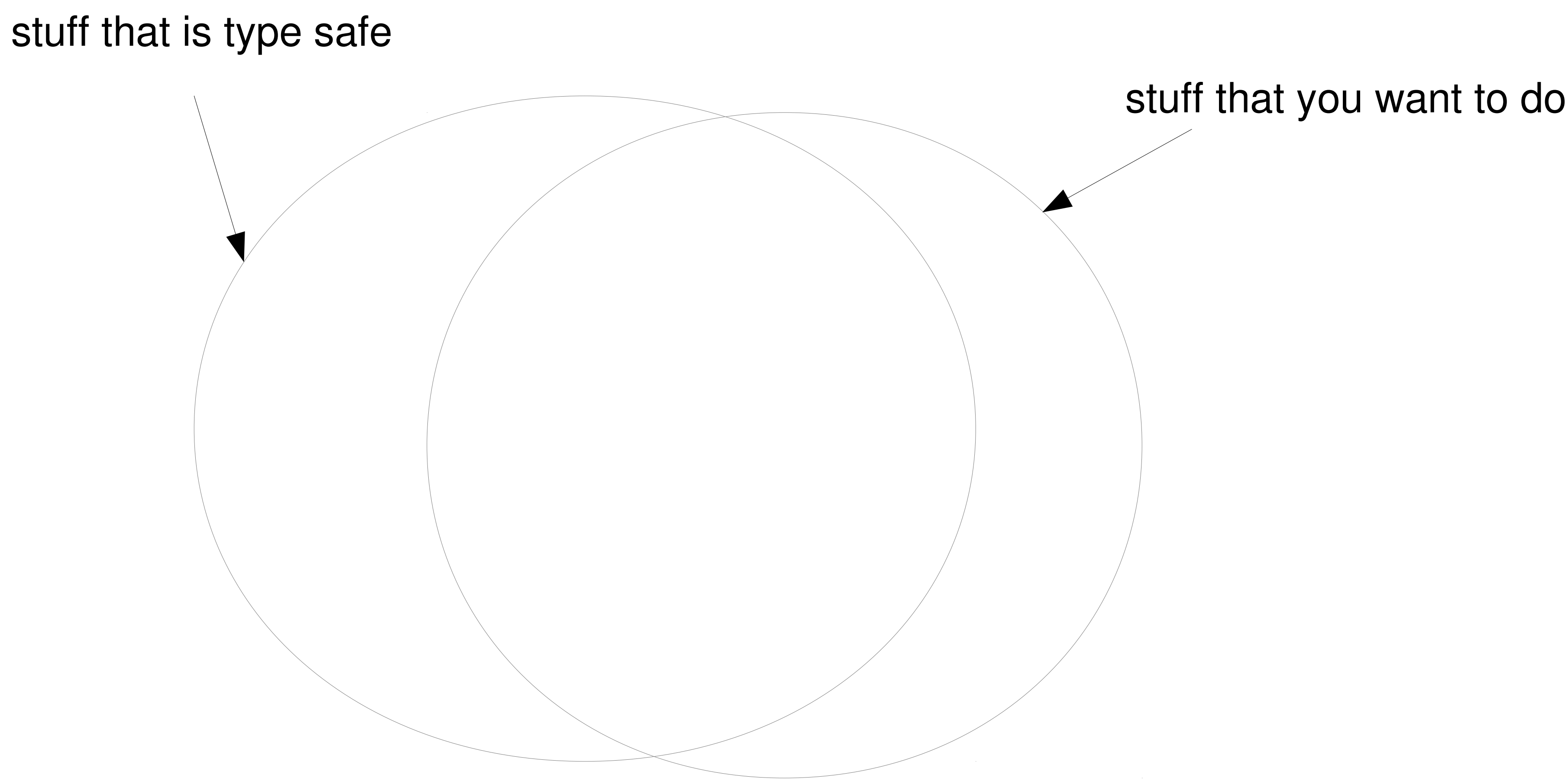
- functions are values (with types)
  - higher order
- no mutable state by default
  - new values
  - names can be used once in a scope
- lazy evaluation
  - “nothing” computed until needed

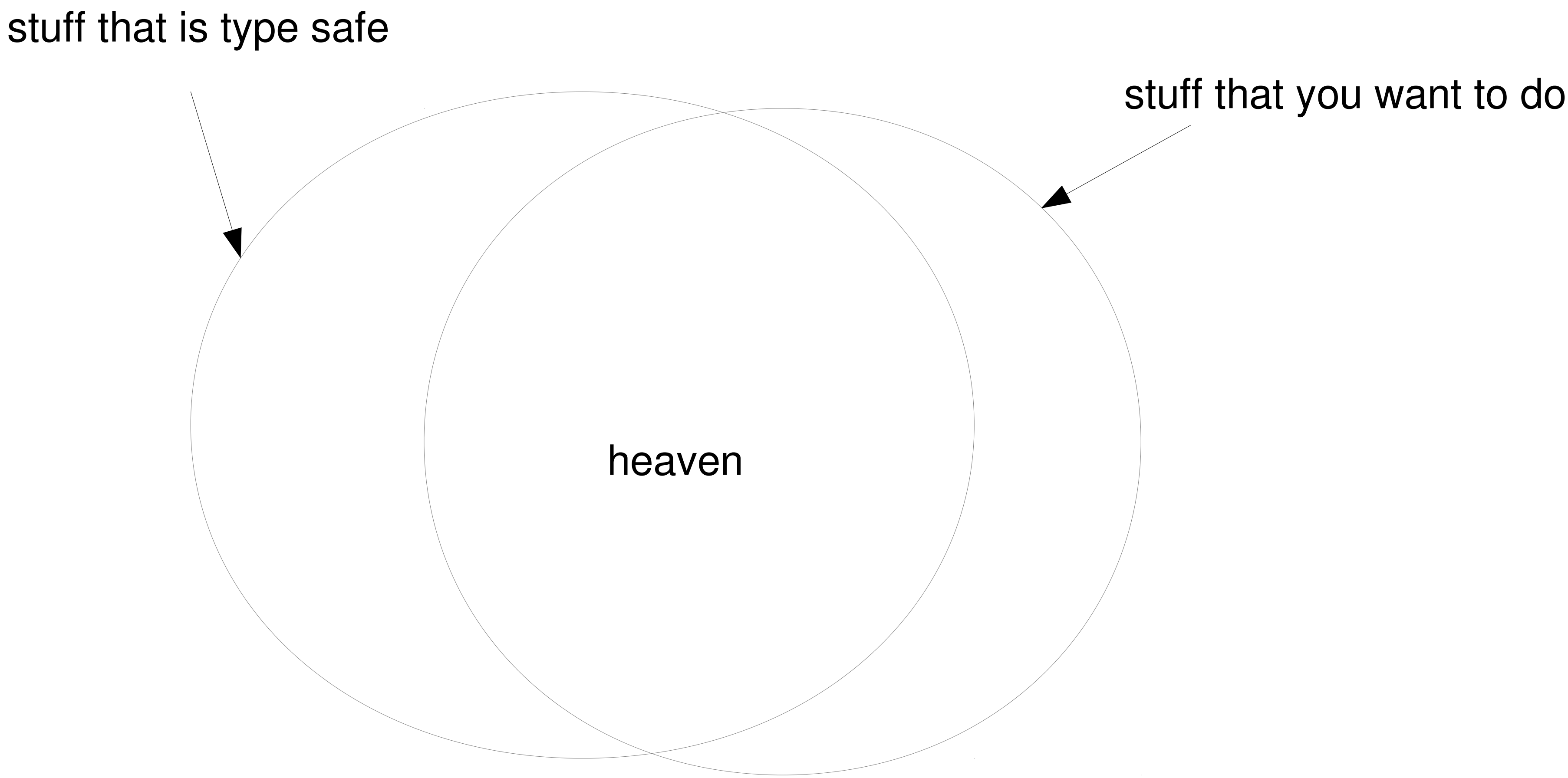
separate out calculations from side effects

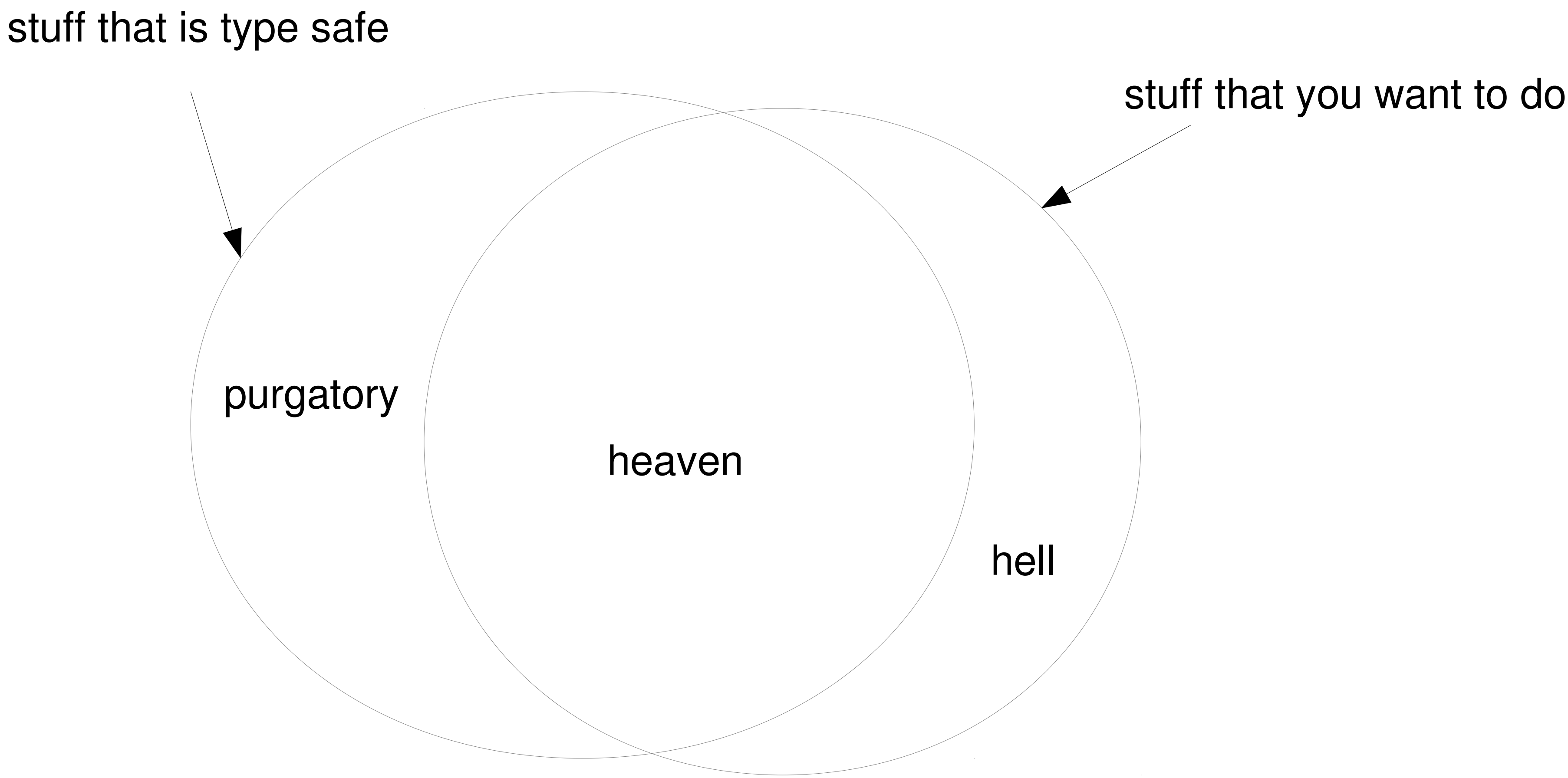
Programming in Haskell is like having a logician on your shoulder.

Programming in Haskell is like having a logician on your shoulder.

– me


stuff that is type safe
stuff that you want to do


stuff that is type safe
stuff that you want to do
heaven


stuff that is type safe
stuff that you want to do
purgatory
heaven
hell

http://www.flickr.com/photos/60414609@N00/5298363220/

constants

```
a = 10
pi = 3.14159
name = “Eric”
pisquared = pi * pi
```

great for fib

```
fib 0 = 1

fib 1 = 1

fib n = fib (n-1) + fib (n-2)
```

lists

```
map _ []      = []
map f (x:xs) = f x : map f xs

concat [] l2 = l2
concat (l:ls) l2 = l : concat ls l2
```

types

```
a :: Int
a = 10
```

types

```
a :: Int
a = 10

name :: String
name = “Eric”
```

types

```haskell
a :: Int
a = 10

name :: String
name = “Eric”

fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

types

```
a :: Int
a = 10

name :: String
name = “Eric”

fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

functions all the way down

```
double x = 2 * x
```

functions all the way down

```
double x = 2 * x

double = 2 *
```

functions all the way down

```
double x = 2 * x

double = 2 *


(*) :: Int → Int → Int

(*) :: Int → (Int → Int)
```

functions all the way down

```
double x = 2 * x
double = 2 *


(*) :: Int → Int → Int
(*) :: Int → (Int → Int)


double :: Int → Int
(2 *)  :: Int → Int
```

scoping

- function application
  - left to right

```
f g h 1 ==> (((f g) h) 1)
```

```
map map square [[1,2],[3,4]]
    ==> (((map map) square) [[1,2],[3,4]])
```

```
map (map square) [[1,2],[3,4]]
```

take a break

we are about to embark
on a journey deep into the type system

http://www.flickr.com/photos/samrich2003/5654034532/

you cannot ignore types

- in Java, you look at types and think “this method takes a string, an int, and returns a list”
  - that's all you need to think about
- in Haskell, this is enough to get you started but you will hit a ceiling

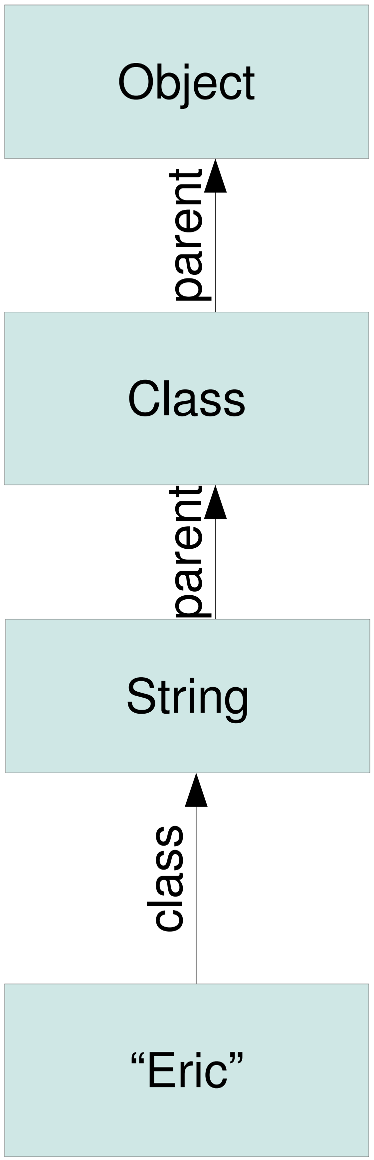
master the types or they will master you

typical dynamic class diagram

objects have a class

classes have a parent

classes are objects

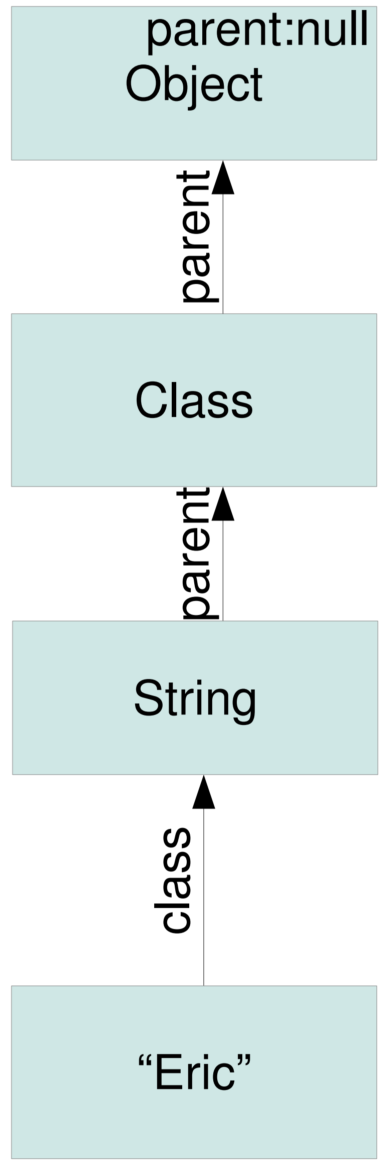


typical dynamic class diagram

objects have a class

classes have a parent

classes are objects

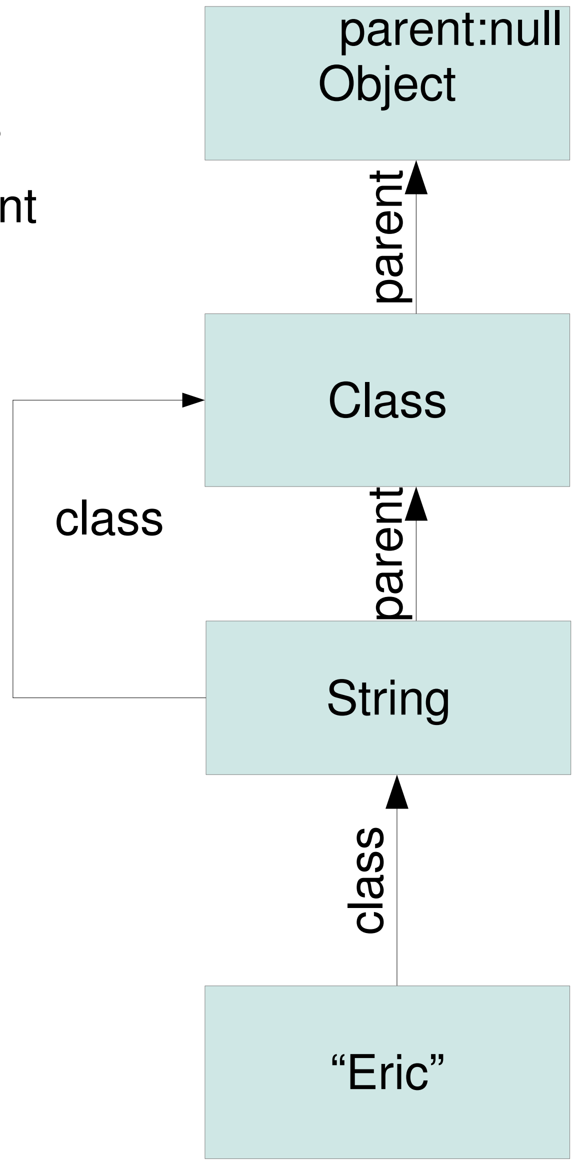


typical dynamic class diagram

objects have a class

classes have a parent

classes are objects

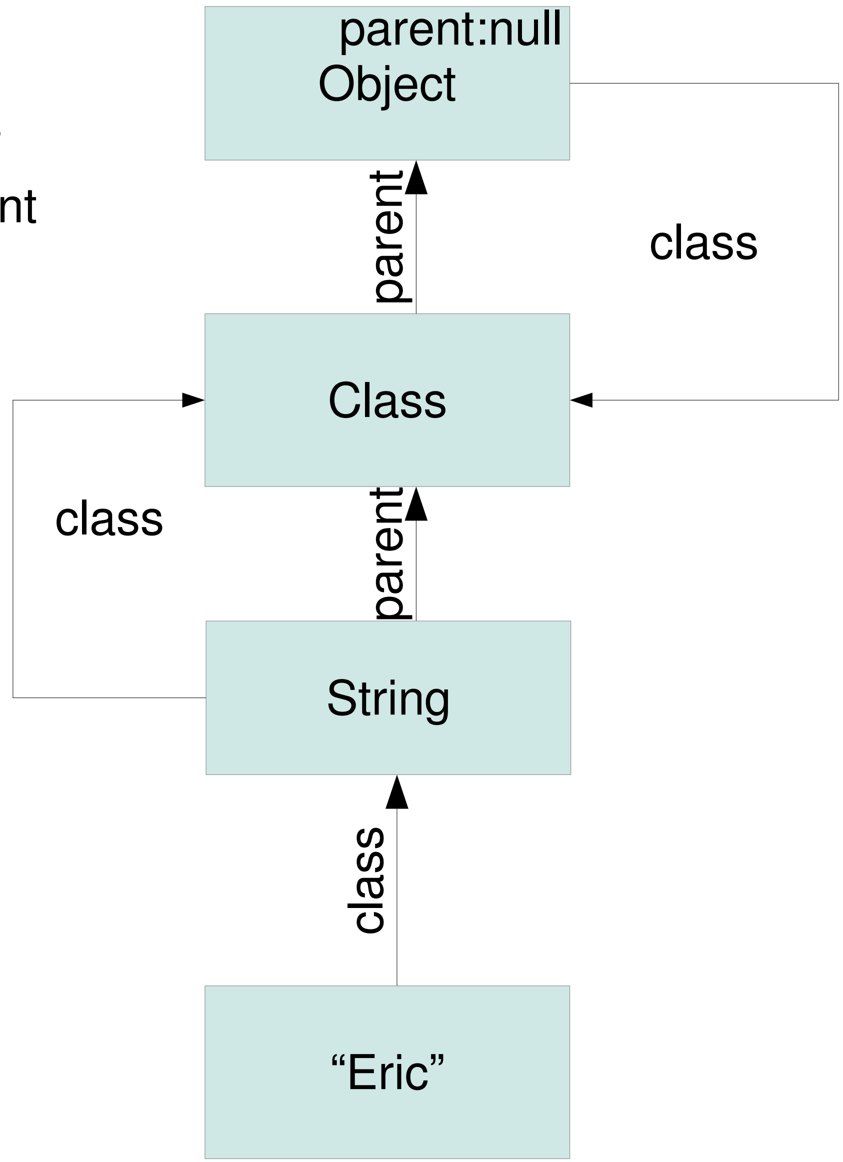


typical dynamic class diagram

objects have a class

classes have a parent

classes are objects

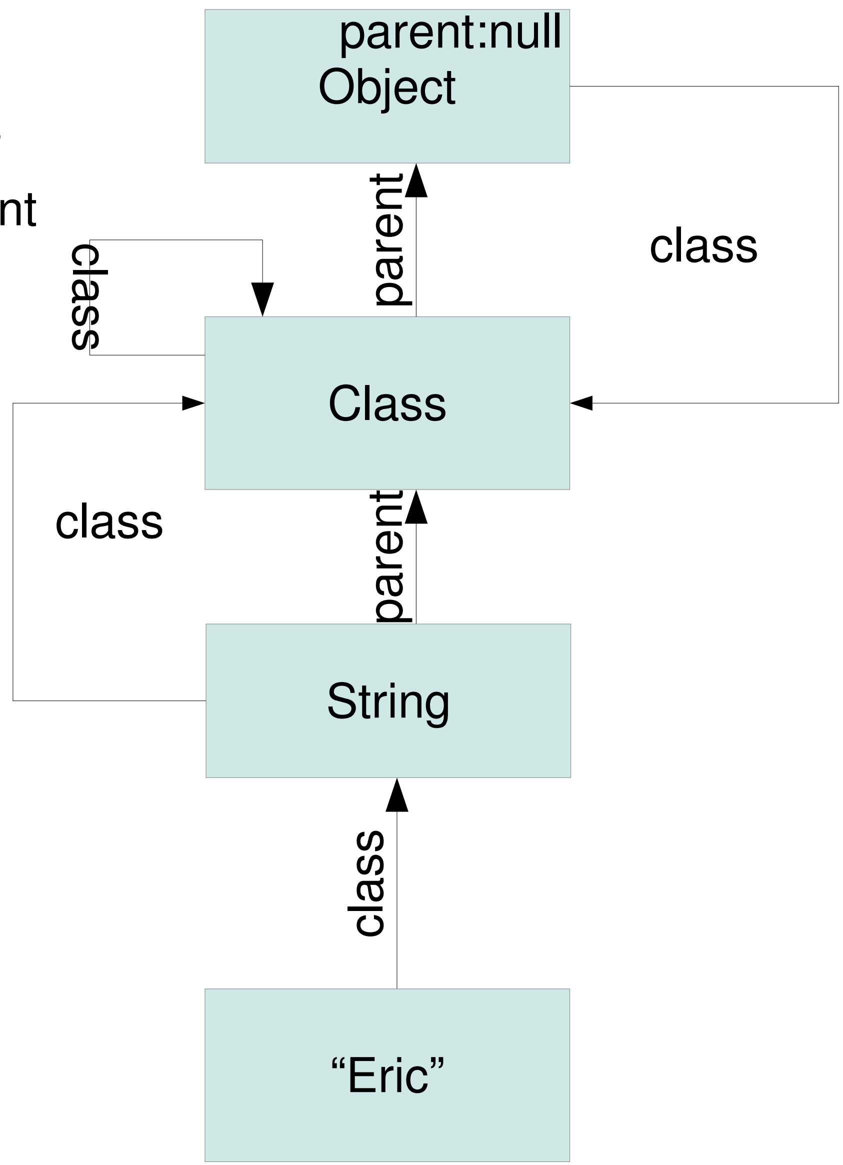


typical dynamic class diagram

objects have a class

classes have a parent

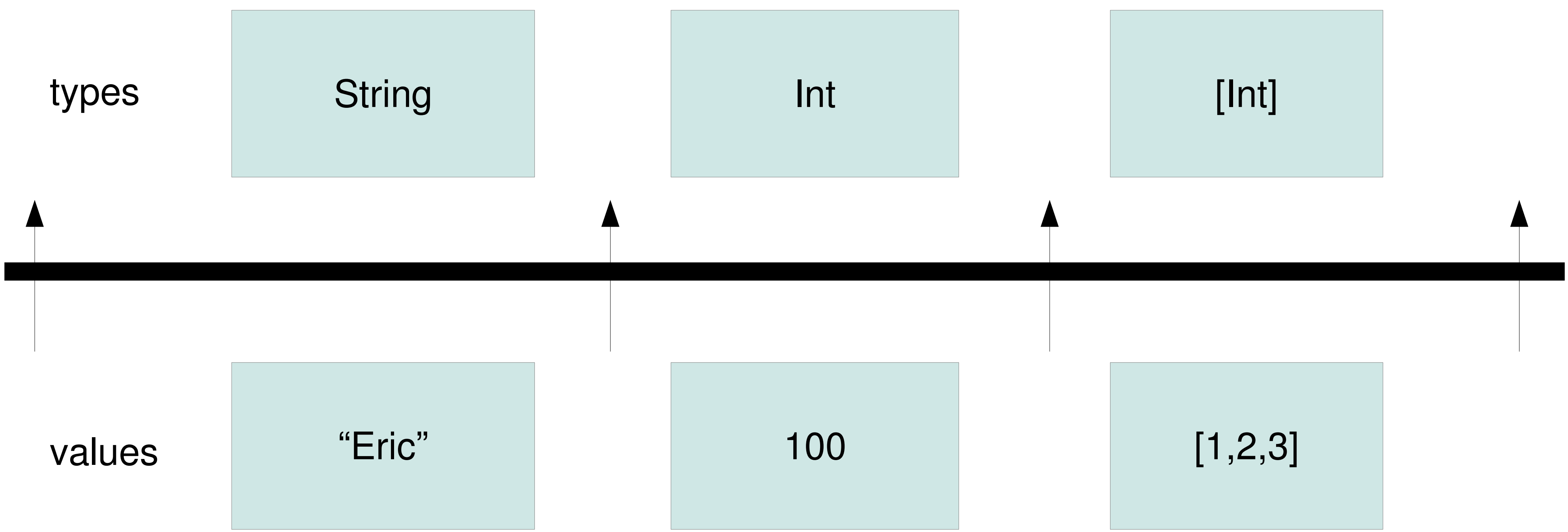
classes are objects



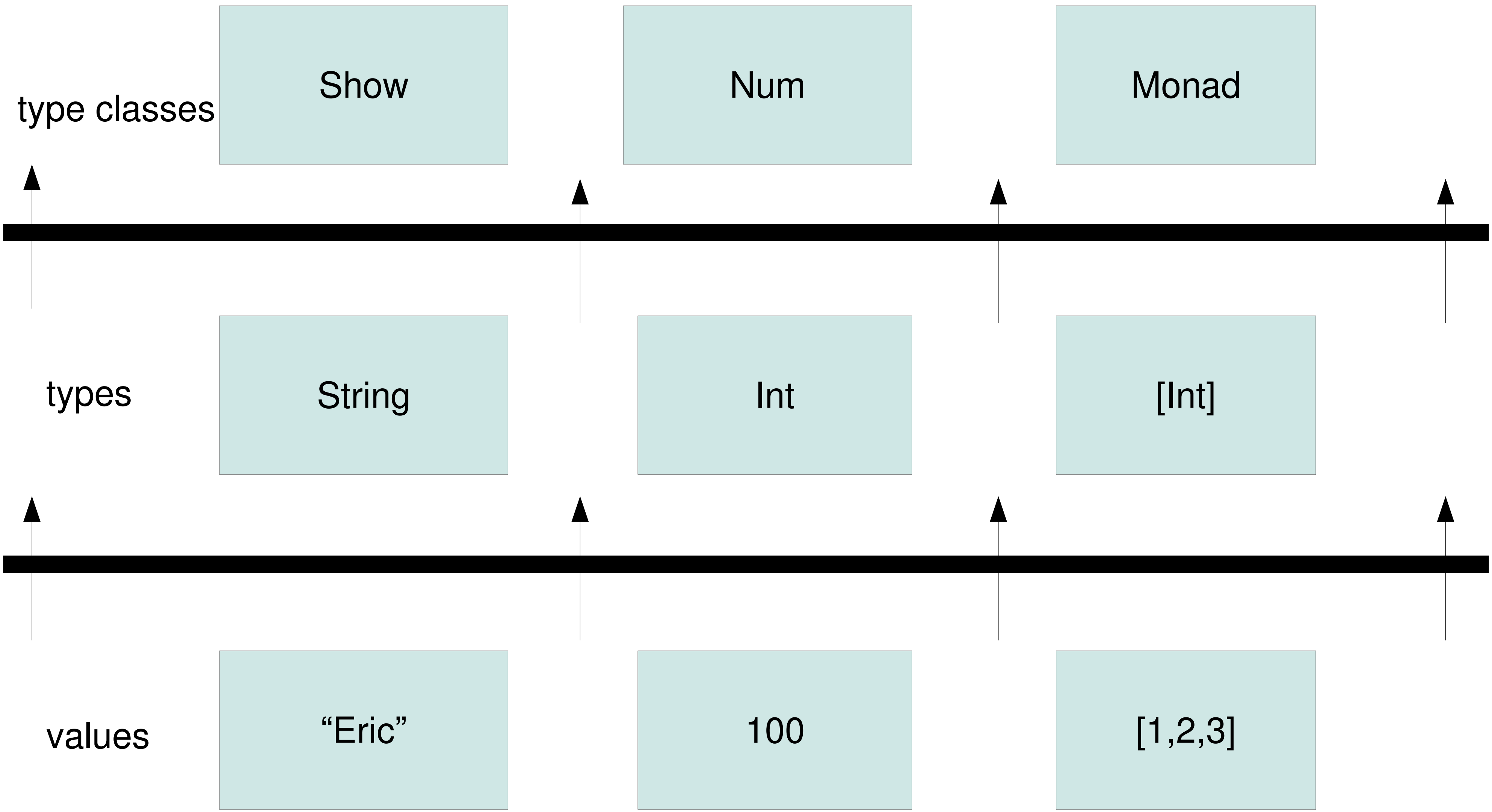
Haskell types

values	“Eric”	100	[1,2,3]

Haskell types



Haskell types



Maybe

- ever thrown a null pointer exception?
- Maybe is the answer

```
data Maybe a = Nothing | Just a
```

Maybe

- ever thrown a null pointer exception?
- Maybe is the answer

```
data Maybe a = Nothing | Just a

find :: (a → Bool) → [a] → Maybe a
```

Maybe

- ever thrown a null pointer exception?
- Maybe is the answer

```
data Maybe a = Nothing | Just a
find :: (a → Bool) → [a] → Maybe a
find _ [] = Nothing
```

Maybe

- ever thrown a null pointer exception?
- Maybe is the answer

```
data Maybe a = Nothing | Just a
find :: (a → Bool) → [a] → Maybe a
find _ [] = Nothing
find p (x:_) | p x = Just x
```

Maybe

- ever thrown a null pointer exception?
- Maybe is the answer

```
data Maybe a = Nothing | Just a
find :: (a → Bool) → [a] → Maybe a
find _ [] = Nothing
find p (x:_) | p x = Just x
find p (_:xs) = find p xs
```

a little safety

```
case find even [1,3,5,7] of
    Just n    → …
    Nothing → ...
```

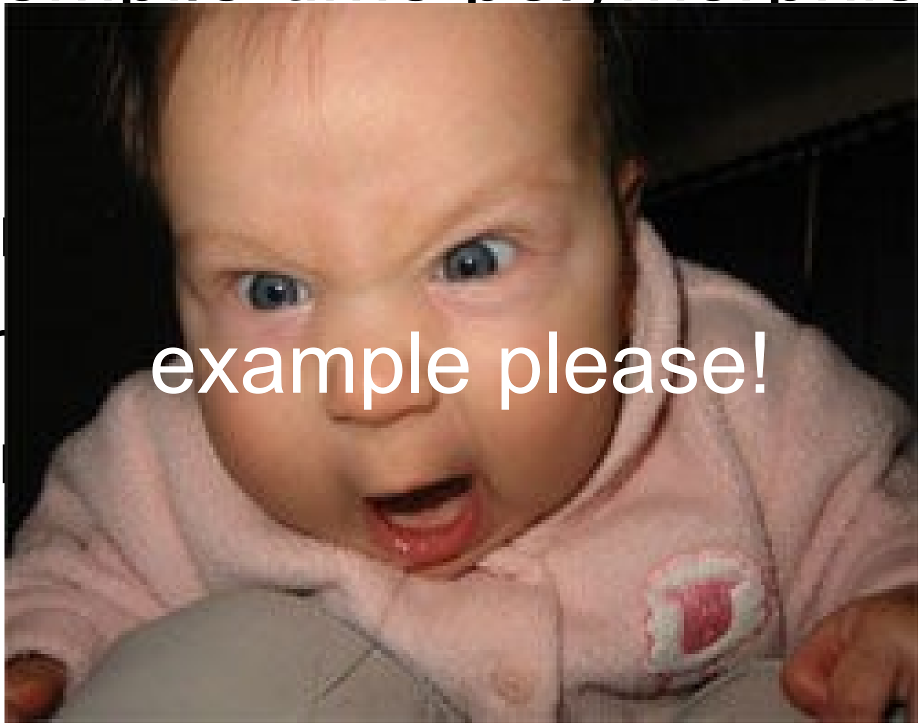
- compiler will complain if you patter match and forget to check for all cases
- but there are still holes
  - fromJust :: Maybe a → a
  - errors if it's Nothing

type classes

- used for compile-time polymorphism
- huh?
  - they define an interface
  - set of functions
  - implementations for a given type

type classes

- used for compile-time polymorphism
- huh?
  - they defi
  - set of fur
  - implemer



http://www.flickr.com/photos/24853457@N00/10675300/

type class Show

```
class Show a where
  show :: a → String

instance Show String where
  show s = s

instance Show Int where
  show = intToString
```

type class Num

```
class Num a where
   (+) :: a → a → a
   (*) :: a → a → a
   (-) :: a → a → a
   (/) :: a → a → a

instance Num Int where
   (+) = intPlus
   (*) = intTimes
   (-) = intMinus
   (/) = intDivide
```

Monad

- don't panic
- not that hard
- a way to compose actions
- type class
  - with type parameter
- IO is a special Monadic type handed down from the gods (the Haskell compiler/runtime)

Monad

- don't panic
- not that ha
- a way to co
- IO is a spe wn from the gods

Monad type class

```
class Monad m where
  -- bind (or then)
  (>>=) :: m a -> (a -> m b) -> m b
  -- create a new value in the Monad
  return :: a -> m a
```

Maybe is a Monad

- Maybe is common. You don't want to indent every time you need to check for Nothing.
- Monads can help.

```
instance Monad Maybe where
  Nothing  >>= _ = Nothing
  (Just a) >>= f = f a
  return = Just
```

using our Monad

```
readint :: String → Maybe Int
index :: [a] → Int → Maybe a


readint s >>= index [1,2,3]
```

dev environment

- Haskell Platform
- emacs (or any text editor)
  - haskell-mode
    - insert type signatures
    - interactive shell
- vim has syntax highlighting
  - haskellmode
- apt-get install cabal-install

books and tutorials

- Real World Haskell
  - free online
- Learn you a Haskell for Great Good
  - free online
- I cannot recommend a Monad tutorial
- The Monad Reader (newsletter)

resources

- hoogle : search engine for Haskell libraries
  - understands types and special characters
- hackage : Haskell library repository
  - versioned libraries with dependencies
  - wild west

final tips

- Haskell is used for two things
  - programming
  - theorem proving
- More blog posts are written for the second one
- Compile often; make the compiler your friend
- Learn the standard libraries
- Model your problem in types