# Stack

- if installed, stack new hello simple
- stack build
- stack exec hello

# Types

- All values have a type.
- All functions have a type.

# Sum Types

```haskell
data Bool = False | True

data Maybe a = Nothing | Just a

data Either a b = Left a | Right b

data [] a = [] | a : [a]
```

# Products

```haskell
data (,) a b = (,) a b

data (->) a b
```

# Sum type

A disjunction (“or”) of possible values. The cardinality is additive.

# Product type

A conjunction of two types. The cardinality is multiplicative.

# Typeclasses

Very briefly, typeclasses are records of functions.

# Class and instance

A `class` declaration gives type signatures for the functions or values of that class. The instance defines how each function or value is implemented for that particular type. The pairing should be unique.

# Eq class

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

# Concrete types

- `function :: Integer -> Integer -> Integer`
- What could this function be? What do we know about `Integer`?

# Polymorphism

- Polymorphic type variables allow us to implement expressions without knowing what *type* the argument will be.
- Polymorphic functions can accept arguments of different types.

# Polymorphism “stack”

- Parametric: type could be anything
- Parametricity means the inputs cannot change the behavior of the function.
- Ad hoc or constrained: constrained by a typeclass

# Typeclass constraints

- define implementations of operations;
- define a unique relationship between type and typeclass instance.
- When the compiler sees values of concrete types, it knows which function implementation to use based on those typeclass instances.

# Polymorphism continued

- `function :: Integer -> Integer -> Integer`

  Permits arguments of only one type.

- `function :: Num a => a -> a -> a`

  Permits arguments of several types (`Integer`, `Int`, `Float`, etc.).

- `function :: a -> a -> a`

  Now `a` is fully polymorphic and can be *any* type.

- *a* equals *a* throughout a type signature.

# Practice:

```
[a] -> Int -> a
```

- returns one element of type *a* from a list
- returns an `Int` value

# What will happen?

For the function on the last slide, what will happen if there isn't a value at the indexed location?

# THIS

*** Exception: Your father smelt of elderberries.

- It's a partial function. It's in Prelude. No, we don't like it.

# Practice:

```
function :: [[a]] -> [a]
```

- takes a list of lists as an argument
- returns a list of functions

# Ad hoc polymorphism

- The typeclass constraint gives you some information about a type.
- Limits the number of types but increases the defined operations you can use.
- So there are more functions you can implement!

# Types and terms

- inverse relationship
- The more types it could possibly be, the fewer implementations there are.

# You can go from polymorphic to concrete

- but you can't go back again!

# TO THE REPL!

General -> Specific
Specific -> General?

# Write your own instance!

```
data Animals = Dog | Cat
instance Eq Animals where
    (==) Dog Dog = True
    (==) Cat Cat = True
    (==) _ _     = False
```

# A more exciting instance

```
data Seuss =  Seuss Colors Animals deriving Show

instance Eq Seuss where
    (==) (Seuss color animal)
         (Seuss color' animal') =
         (color == color') && (animal == animal')
```