Type classes
But what about (+)?

\[
(+) :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}
\]
\[
(+) :: a \rightarrow a \rightarrow a
\]

**ad hoc polymorphism**

the ability for an entity to behave differently on different “input” or “contained” types
**Type class** (in Haskell)

A set of types defined by an interface (set of functions) that the types must implement.

### Example

<table>
<thead>
<tr>
<th>class Eq a where</th>
<th>&gt; :type (==)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(==) :: a -&gt; a -&gt; Bool</td>
<td>(==) :: Eq a =&gt; a -&gt; a -&gt; Bool</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>class Show a where</th>
<th>&gt; :type show</th>
</tr>
</thead>
<tbody>
<tr>
<td>show :: a -&gt; String</td>
<td>show :: Show a =&gt; a -&gt; String</td>
</tr>
</tbody>
</table>

Eq is a **type class**.

Eq a => is called a **type class constraint** on (=).
Polymorphic values revisited

[] :: [a]
undefined :: a

1 :: Num a => a
“Higher-order” type classes

But what about map?

\[
\text{listMap ::} \\
(a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

\[
\text{streamMap ::} \\
(a \rightarrow b) \rightarrow \text{Stream } a \rightarrow \text{Stream } b
\]

\[
\text{vectorMap ::} \\
(a \rightarrow b) \rightarrow \text{Vector } a \rightarrow \text{Vector } b
\]
But what about `map`?

```haskell
listMap :: (a -> b) -> [a] -> [b]

streamMap :: (a -> b) -> Stream a -> Stream b

vectorMap :: (a -> b) -> Vector a -> Vector b
```

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

instance Functor [] where
  -- fmap :: (a -> b) -> [a] -> [b]
  fmap = map
```
Map is generic in two type variables

```
data Map k v = ...
```

Map \( k \) (curried!) is an instance of Functor:

```
instance Functor (Map k) where
    fmap :: (a -> b) -> Map k a -> Map k b
    fmap = ...
```

Demo!

Modeling mutation in a pure functional world
Hopefully your work in this course up to this point has convinced you that explicit mutation is not necessary to write substantial programs!

But sometimes a domain/algorithm is most easily modeled using mutable state.

Problem: given a recursive function, count the total number of times the function is called.

\[
\begin{align*}
\text{fib} \ 0 &= 0 \\
\text{fib} \ 1 &= 1 \\
\text{fib} \ n &= \text{fib} \ (n-1) + \text{fib} \ (n-2)
\end{align*}
\]
Python solution

count = 0

def fib_counted(n):
    global count
    count += 1
    if n < 2:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
The Python code uses a global mutable counter to keep track of the number of “calls so far”.

How do we do this without using mutation?

Recall foldl’s update function

```
update :: acc -> x -> acc
```

We can add an extra parameter and return value to \( \text{fib} \) to “accumulate” a counter.
In most languages, mutable state is **implicit**, managed by the language implementation.

In pure functional programming, we make the “mutable” state **explicit** in a function’s input/output.
In general, if function $f :: t_1 \rightarrow \ldots \rightarrow t_n \rightarrow a$ uses mutable state of type $s$, we model this as a function

$$f' :: t_1 \rightarrow \ldots \rightarrow t_n \rightarrow s \rightarrow (a, s)$$

Haskell’s built-in `State` type constructor models a stateful operation

```hs
data State s a = State (s -> (a, s))
```

- $s$: type of the “mutable” state
- $a$: type of the “return” value of the operation
- $s \rightarrow (a, s)$: takes old state, returns a value and new state
Primitive state operations

\[
\begin{array}{l}
\text{get :: State } s \rightarrow s \\
\text{get = State } (\lambda \text{state} \rightarrow (\text{state, state})) \\
\text{put :: } s \rightarrow \text{State } s () \\
\text{put } x = \text{State } (\lambda \rightarrow ((), x))
\end{array}
\]

() is called “unit”, similar to “void” in C/Java.

Extracting/performing a state operation

\[
\begin{array}{l}
\text{runState :: State } s \text{ a } \rightarrow (s \rightarrow (a, s)) \\
\text{runState } (\text{State } f) = f
\end{array}
\]

-- equivalently,
\[
\begin{array}{l}
\text{runState :: State } s \text{ a } \rightarrow s \rightarrow (a, s) \\
\text{runState } (\text{State } f) \text{ init } = f \text{ init}
\end{array}
\]
Demo: using State with fibCounted

One subtlety: the built-in Haskell State doesn’t make public the State value constructor. Instead, use the function

```
state :: (s -> (a, s)) -> State s a
```
Chaining stateful operations

Problem: too much boilerplate code

```
fibCounted n = State $ \count \rightarrow
  let (f1, count1) = runState (fibCounted (n - 1)) count
      (f2, count2) = runState (fibCounted (n - 2)) count1
  in
      (f1 + f2, count2 + 1)
```
Problem: too much boilerplate code

```
fibCounted n = State $ \count ->
  let (f1, count1) = runState (fibCounted (n - 1)) count
      (f2, count2) = runState (fibCounted (n - 2)) count1
  in (f1 + f2, count2 + 1)
```

Sequencing attempt 1

Want to elegantly sequence stateful operations: “perform op1, and then op2.”

```
andThen :: State s a -> State s b -> State s b
andThen stateOp1 stateOp2 = State $ \state0 ->
  let (x1, state1) = runState stateOp1 state0
      (x2, state2) = runState stateOp2 state1
  in (x2, state2)
```
Sequencing attempt 2

Want to elegantly sequence stateful operations: “perform op1, and then op2 with the result of op1 in scope.”

```haskell
bind :: State s a -> (a -> State s b) -> State s b
bind stateOp1 stateOp2 = State $ \state0 ->
    let (x1, state1) = runState stateOp1 state0
        (x2, state2) = runState (stateOp2 x1) state1
    in
        (x2, state2)
```

Using bind with fibCounted
Generalizing state

```haskell
bind :: State s a -> (a -> State s b) -> State s b

class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
```