Return type polymorphism
Recall type classes

In Haskell, a function defined in a type class has multiple implementations. The compiler selects the right one when the argument type is inferred.

```haskell
x == y
f x >>= \y -> return [y]
```

Java's method overloading

This seems more general than type classes!

```java
class Point {
    void move(int dx, int dy) { ... }
    int move(float dx, float dy) { ... }
    String move() { ... }
}
Point p = Point();
p.move(1, 2);
p.move(1.5, 3.5);
p.move();
```
Now consider Read:

```haskell
class Read a where
    read :: String -> a
```

`read` is ad-hoc polymorphic in its return type.

This is known as **return type polymorphism**.

This isn’t allowed by Java!

```java
class Point {
    void   move(int dx, int dy) { ... }
    int    move(int dx, int dy) { ... }
    String move(int dx, int dy) { ... }
}
```
When `move` is called in Java, the compiler chooses the right implementation based on the *arguments*, not its return type. Why?

Because of side effects, the return value of a function can be ignored—the compiler might never see it!

```java
Point p = Point(2, 3);
p.move(10, 20);
p.move(1, 2);
p.move(100, 0);
```
Contrast this with Haskell:

```haskell
read :: Read a => String -> a

take (read "10") [1..100]
True && (read "False")
True && head (read "[True, False]")
```

Or consider return :: Monad a => a -> m a

```haskell
f :: Maybe String -> State Integer String -> IO String
   -> ...
f = ...

f (return "David") (return "David") (return "David")
```
But type inference isn’t magic! Consider this function:

\[
\begin{aligned}
f &:: \text{String} \to \text{String} \\
f \text{input} &= \text{show} \ (\text{read} \ \text{input})
\end{aligned}
\]

Combining monads
We’ve seen two monads representing different kinds of effects: \texttt{State} \ s (stateful computations), and \texttt{IO} (input/output).

How do we express a computation that combines these effects?

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

But also print out every intermediate count.
**Attempt 1**

The a is now an IO action:

\[
\text{fibCountedIO} :: \text{Integer} \rightarrow \text{State}\text{.State Integer (IO Integer)}
\]

**Attempt 2**

Using the \texttt{StateT} monad transformer, which is a “higher-order monad”:

\[
\text{data StateT s m a = StateT (s -> m (a, s))}
\]

Haskell implements the following “monad lifting”:

- If m is a monad, then StateT s m is also a monad.
Note: state, get, and put actually work with all StateT instances.

```haskell
state :: Monad m => (s -> (a, s)) -> StateT s m a
get :: Monad m => StateT s m s
put :: Monad m => s -> StateT s m ()
```