Monads

All your types are belong to us

Ashton Wiersdorf

What is a monad?

monad = interface

```java
public interface Monad<T> {
    Monad<T> wrap(T value);
    <R> Monad<R> thread(Function<T, Monad<R>> f);
}
```

```haskell
class Monad m where
    wrap :: a -> m a
    thread :: m a -> (a -> m b) -> m b
```

`Monad` **is a sub-interface of** `Functor` **and** `Applicative`

Functor

A box you can `map` over

```
class Functor m where
    map :: (Functor f) => (a -> b) -> f a -> f b
```

`Monad` **is a sub-interface of** `Functor` **and** `Applicative`

Applicative

A generalization of a function
Something you can apply

```
class Applicative m where
    wrap  :: (Applicative f) ⇒ a → f a
    apply :: (Applicative f) ⇒ f (a → b) → f a → f b
```

```haskell
class Monad m where
    wrap :: a -> m a
    thread :: m a -> (a -> m b) -> m b
```

Demo: the “Foo” monad that does nothing

Why do we have monads?

Monads are good at modeling effects

What are effects?

What are (side-)effects?

Why do we care about effects, especially in functional programming?

Monads let us encode effects

Example: encoding exceptions

What is a monad *really*?

Interfaces vs Typeclasses

Interfaces

- **Closed**
  Defined with class definition; can’t be added later

- **Requires instance**
  Interfaces require you have an instance to invoke a method on

Typeclasses

- **Open**
  Typeclasses can be implemented anywhere, anytime.

- **Works with return types**
  Typeclasses can dispatch based off of the expected return type (e.g. `pure`)

What if my language doesn’t have typeclasses?

Using monads comfortably

```haskell
do
  x ← Just 42
  y ← Just (x + 1)
  Just (y * 2)
```

```haskell
do
  x ← Just 42
  y ← Just (x + 1)
  Just (y * 2)

Just 42 >>=
  (\x → (Just (x + 1)) >>=
     (\y → Just (y * 2)))
```

Lots of functions beyond the monad interface

Demo: write a linear congruence generator

$$X_{n+1} = (aX_n + c) \mod m$$

Monad laws

```
return x >>= f
```

is the same as

```
f x
```

`x >>= return`

is the same as

`x`

```
x >>= f >>= g
```

is the same as

```
(x >>= f) >>= g
```

is the same as

```
x >>= (\y -> (f y) >>= g)
```