



Meetup



Friendly Environment Policy



Berlin Code of Conduct



**CATEGORY THEORY  
FOR PROGRAMMERS**



Bartosz Milewski

**Category  
Theory  
for  
Programmers**  
**Chapter 23:**  
**Comonads**

<b>23</b>	<b>Comonads</b>	<b>337</b>
23.1	Programming with Comonads . . . . .	338
23.2	The Product Comonad . . . . .	339
23.3	Dissecting the Composition . . . . .	340
23.4	The Stream Comonad . . . . .	343
23.5	Comonad Categorically . . . . .	345
23.6	The Store Comonad . . . . .	348
23.7	Challenges . . . . .	351

```
class Monad m where
```

```
    (=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

```
    return :: a -> m a
```

```
class Monad m where
```

```
    (>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

```
    return :: a -> m a
```

```
class Functor w => Comonad w where
```

```
    (=>=) :: (w a -> b) -> (w b -> c) -> (w a -> c)
```

```
    extract :: w a -> a
```

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

For every monad, instead of defining the fish operator, we may instead define bind. In fact the standard Haskell definition of a monad uses bind:

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

That leads us to the third option for defining a monad:

```
class Functor m => Monad m where
    join :: m (m a) -> m a
    return :: a -> m a
```

```
class Functor w => Comonad w where
  extract :: w a -> a
  duplicate :: w a -> w (w a)
  duplicate = extend id
  extend :: (w a -> b) -> w a -> w b
  extend f = fmap f . duplicate
```







Meetup