Ceci n'est pas une monad tutorial

Who am I?

- Krisztián Pintér, MSc student at ELTE.
- Wrote my BSc thesis in Haskell.
- DISCLAIMER: Not an expert in monads. (Please don't throw things at me.)

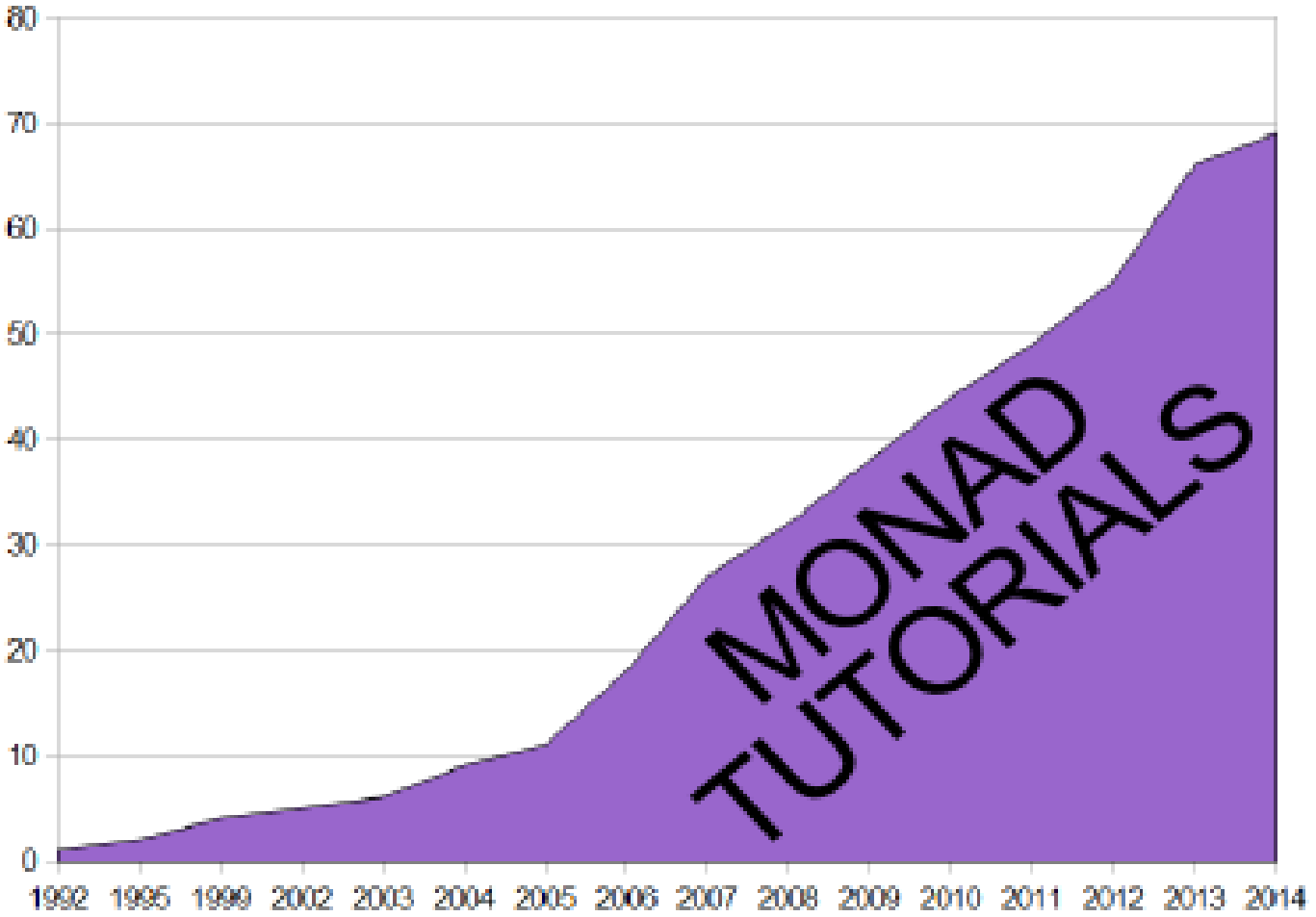
Intention of this presentation

- Give newcomers pointers to understanding monads.
- Give ideas to Haskell veterans to better explain monads.

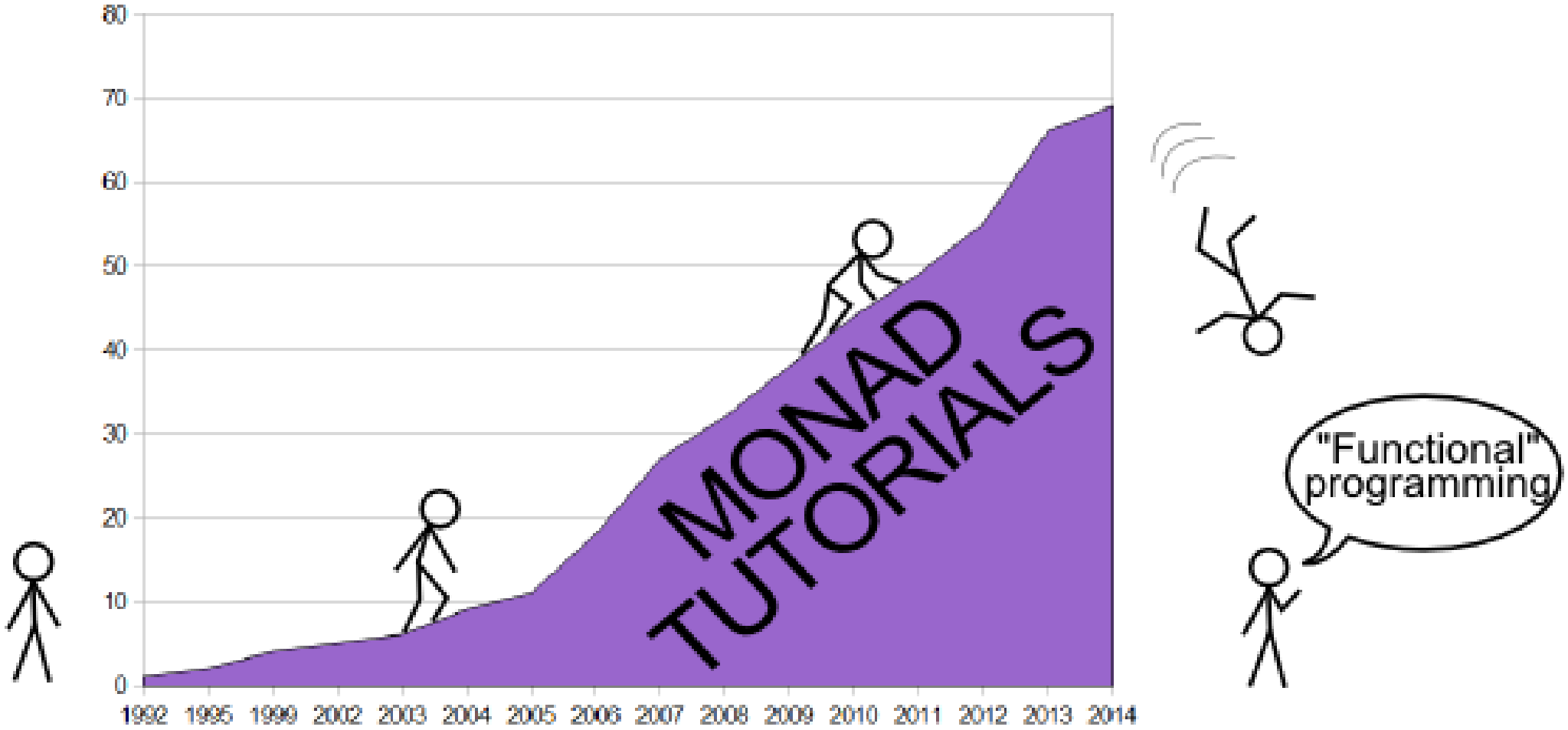
Haskell has an image problem

- People think Haskell is overly complicated
- MONADS?????

Amount of monad tutorials



Amount of monad tutorials



- Many monad tutorials start with definitions
- People feel they have to start with them

But we don't think in definitions...

When you **see a dog**, you know it's a dog because **you've seen dogs before**, not because you think the **definition for "dog"** in your head. (Whatever that is.)

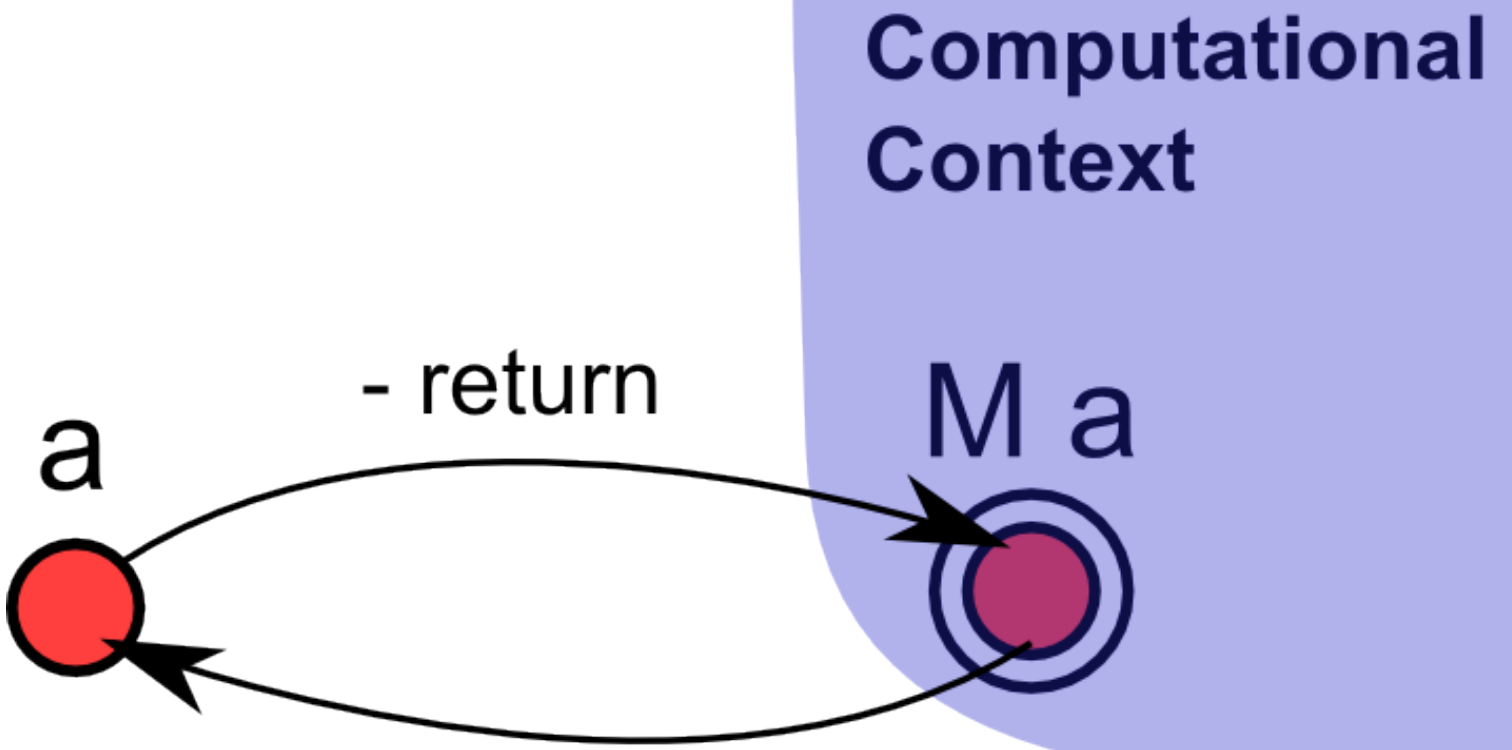
Do definitions help us understand concepts?

- I don't think they necessarily do.
- When you hear a definition, your brain has to imagine fitting examples, to develop intuition.
- Leave definitions after examples.

Monad tutorials

When someone writes a monad tutorial, they usually give you their own interpretation of the very abstract concept of a monad.

My interpretation



- run (State, ST)
- pattern match ([], Maybe)
- IO (not possible*)

Confusing things about monads

A fun game to play

Ask a friend to...

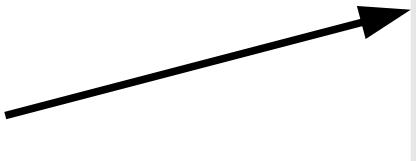
...point at an integer

```
int s,i;
int a[] = {1, 2, 3};

for(i = 0; i < 3; ++i)
{
    s += a[i];
}
```

A fun game to play

Ask a friend to...

...point at an integer



```
int s,i;
int a[] = {1, 2, 3};

for(i = 0; i < 3; ++i)
{
    s += a[i];
}
```

A fun game to play

Ask a friend to...

...point at an integer

...point at a for loop

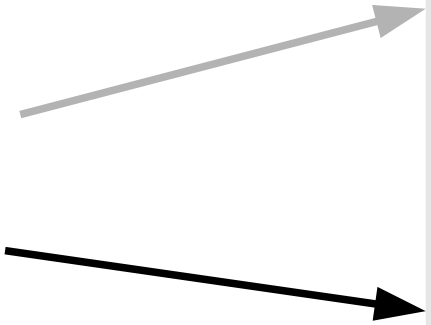
```
int s,i;
int a[] = {1, 2, 3};

for(i = 0; i < 3; ++i)
{
    s += a[i];
}
```

A fun game to play

Ask a friend to...

...point at an integer

...point at a for loop



```
int s,i;
int a[] = {1, 2, 3};

for(i = 0; i < 3; ++i)
{
    s += a[i];
}
```

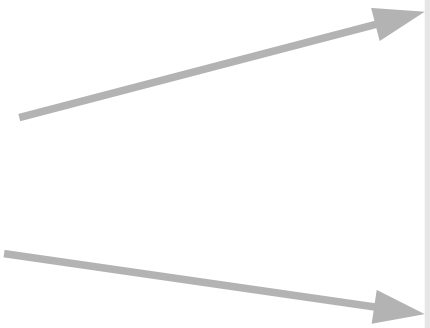
A fun game to play

Ask a friend to...

...point at an integer

...point at a for loop

...point at a mediator pattern



```
int s,i;
int a[] = {1, 2, 3};

for(i = 0; i < 3; ++i)
{
    s += a[i];
}
```

A fun game to play

Ask a friend to...

...point at an integer

...point at a for loop

...point at a mediator pattern

Um...

```
int s,i;
int a[] = {1, 2, 3};

for(i = 0; i < 3; ++i)
{
    s += a[i];
}
```

A fun game to play

```
using System;
using System.Collections;
  class MainApp
  {
    static void Main()
    {
      ConcreteMediator m = new ConcreteMediator();
      ConcreteColleague1 c1 = new
ConcreteColleague1(m);
      ConcreteColleague2 c2 = new
ConcreteColleague2(m);
      m.Colleague1 = c1;
      m.Colleague2 = c2;
      c1.Send("How are you?");
      c2.Send("Fine, thanks");
      // Wait for user
      Console.Read();
    }
  }
```

Ask a friend to...

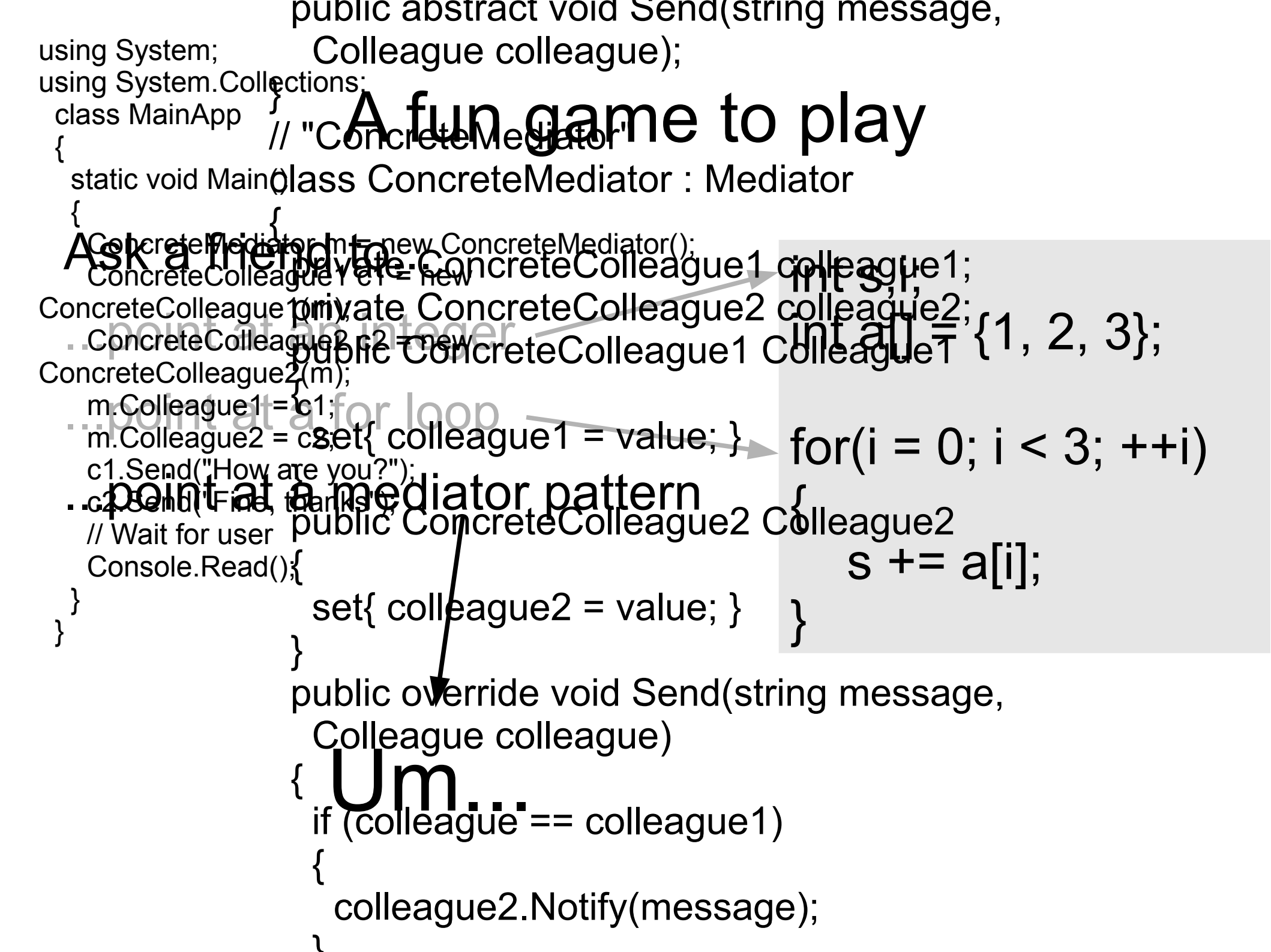
...point at an integer

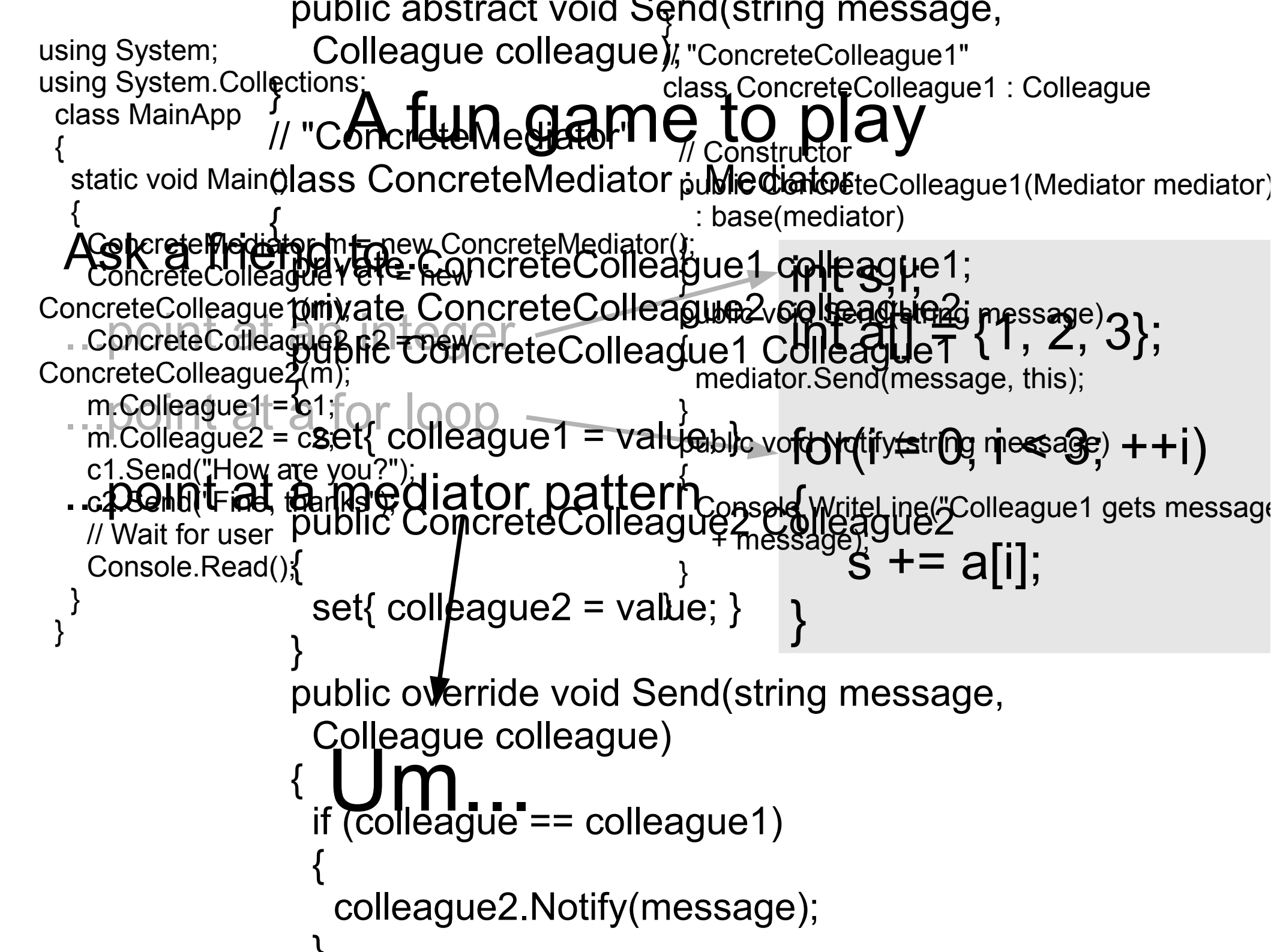
...point at a for loop

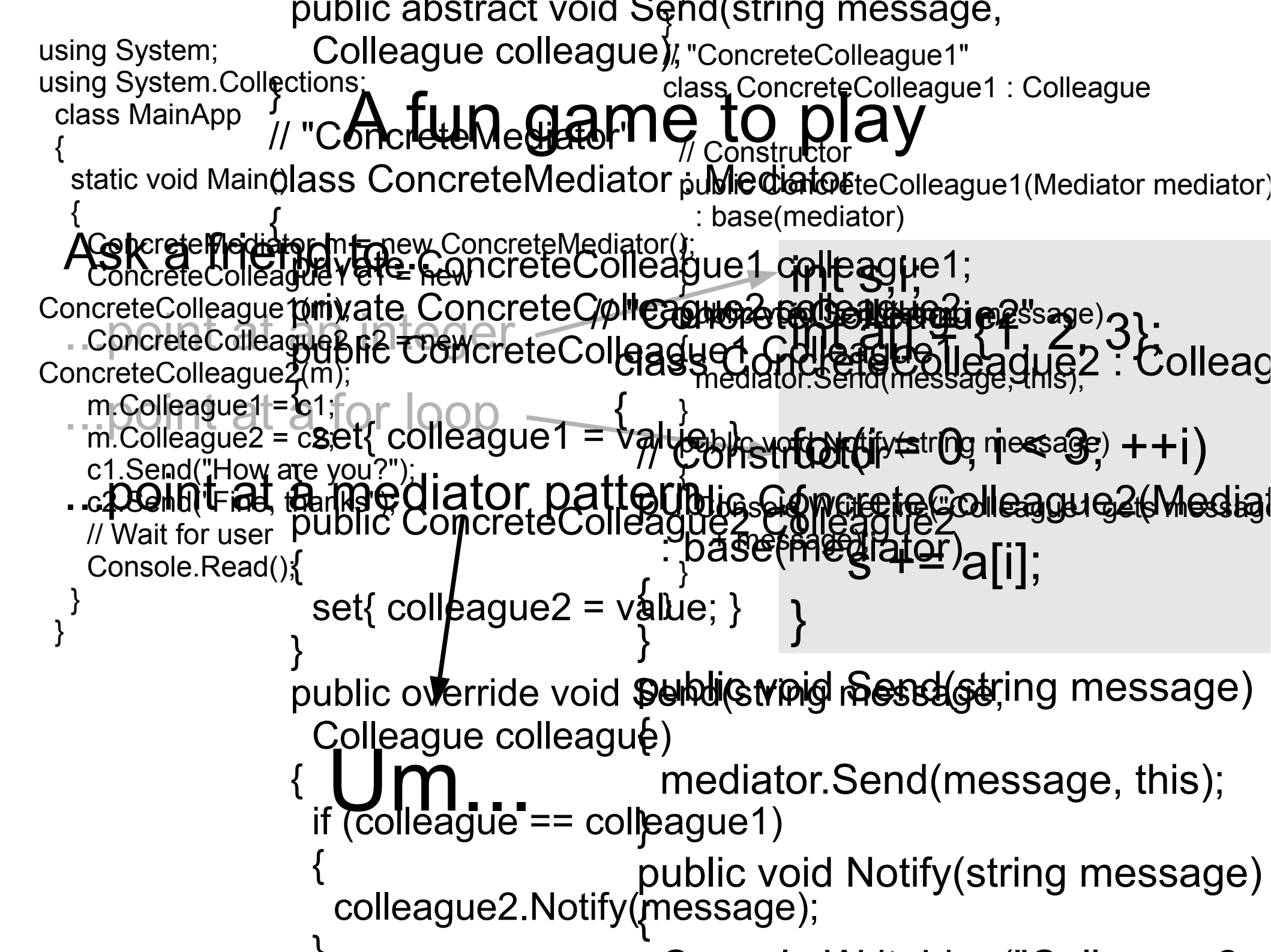
...point at a mediator pattern

```
int s,i;
int a[] = {1, 2, 3};

for(i = 0; i < 3; ++i)
{
    s += a[i];
}
```

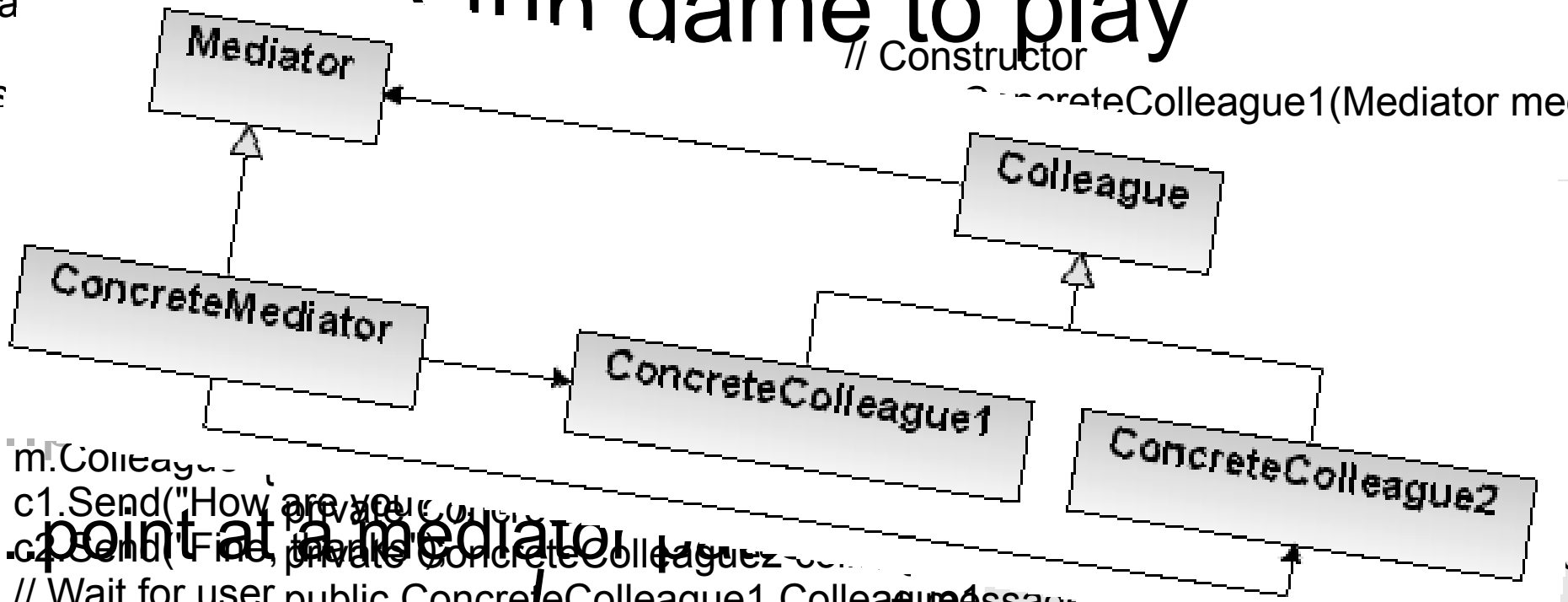
Um...


public abstract void Send(string message,
Colleague colleague);
}
// "ConcreteMediator"
class ConcreteMediator : Mediator
{
private ConcreteColleague1 colleague1;
private ConcreteColleague2 colleague2;
public ConcreteColleague1 Colleague1
{
set{ colleague1 = value; }
}
public ConcreteColleague2 Colleague2
{
set{ colleague2 = value; }
}
public override void Send(string message,
Colleague colleague)
{
if (colleague == colleague1)
{
colleague2.Notify(message);
}
using System;
using System.Collections;
class MainApp
{
static void Main()
{
ConcreteMediator m = new ConcreteMediator();
ConcreteColleague1 c1 = new
ConcreteColleague1(m);
ConcreteColleague2 c2 = new
ConcreteColleague2(m);
m.Colleague1 = c1;
m.Colleague2 = c2;
c1.Send("How are you?");
c2.Send("Fine, thanks");
// Wait for user
Console.Read();
}
}
A fun game to play
Ask a friend to...
...point at an integer
...point at a for loop
...point at a mediator pattern
Um...
int s,i;
int a[] = {1, 2, 3};
for(i = 0; i < 3; ++i)
{
s += a[i];
}


A fun game to play
Ask a friend to...
...point at an integer
...point at a for loop
...point at a mediator pattern
Um...
int s,i;
int a[] = {1, 2, 3};
for(i = 0; i < 3; ++i)
{
s += a[i];
}
using System;
using System.Collections;
class MainApp
{
static void Main()
{
ConcreteMediator m = new ConcreteMediator();
ConcreteColleague1 c1 = new
ConcreteColleague1(m);
ConcreteColleague2 c2 = new
ConcreteColleague2(m);
m.Colleague1 = c1;
m.Colleague2 = c2;
c1.Send("How are you?");
c2.Send("Fine, thanks");
// Wait for user
Console.Read();
}
}
public abstract void Send(string message,
Colleague colleague);
}
// "ConcreteMediator"
class ConcreteMediator : Mediator
{
private ConcreteColleague1 colleague1;
private ConcreteColleague2 colleague2;
public ConcreteColleague1 Colleague1
{
set{ colleague1 = value; }
}
public ConcreteColleague2 Colleague2
{
set{ colleague2 = value; }
}
public override void Send(string message,
Colleague colleague)
{
if (colleague == colleague1)
{
colleague2.Notify(message);
}
// "ConcreteColleague1"
class ConcreteColleague1 : Colleague
{
// Constructor
public ConcreteColleague1(Mediator mediator)
: base(mediator)
{
}
public void Send(string message)
{
mediator.Send(message, this);
}
public void Notify(string message)
{
Console.WriteLine("Colleague1 gets message
+ message);
}
}


A fun game to play
Ask a friend to...
...point at an integer
...point at a for loop
...point at a mediator pattern
Um...
public abstract void Send(string message,
Colleague colleague);
}
// "ConcreteMediator"
class ConcreteMediator : Mediator
{
private ConcreteColleague1 colleague1;
private ConcreteColleague2 colleague2;
public ConcreteColleague1 Colleague1
{
set{ colleague1 = value; }
}
public ConcreteColleague2 Colleague2
{
set{ colleague2 = value; }
}
public override void Send(string message,
Colleague colleague)
{
if (colleague == colleague1)
{
colleague2.Notify(message);
}
// "ConcreteColleague1"
class ConcreteColleague1 : Colleague
{
// Constructor
public ConcreteColleague1(Mediator mediator)
: base(mediator)
{
}
public void Send(string message)
{
mediator.Send(message, this);
}
public void Notify(string message)
{
Console.WriteLine("Colleague1 gets message:
}
}
// "ConcreteColleague2"
class ConcreteColleague2 : Colleag
{
// Constructor
public ConcreteColleague2(Mediat
: base(mediator)
{
}
public void Send(string message)
{
mediator.Send(message, this);
}
public void Notify(string message)
{
int s;
int[] a = { 1, 2, 3};
for (i = 0; i < 3; ++i)
s += a[i];
using System;
using System.Collections;
class MainApp
{
static void Main()
{
ConcreteMediator m = new ConcreteMediator();
ConcreteColleague1 c1 = new
ConcreteColleague1(m);
ConcreteColleague2 c2 = new
ConcreteColleague2(m);
m.Colleague1 = c1;
m.Colleague2 = c2;
c1.Send("How are you?");
c2.Send("Fine, thanks");
// Wait for user
Console.Read();
}
}

A fun game to play



point at a mediator...

Um...

```
using System;
using System.Collections;
cla
{
  s
  /
  C
  C
    m.Colleague
    c1.Send("How are you?");
    c2.Send("Fine, thanks");
    // Wait for user
    Console.Read();
  }
}
```

```
    private ConcreteColleague2 colleague2;

    public ConcreteColleague1 Colleague1
    {
      set{ colleague1 = value; }
    }

    public ConcreteColleague2 Colleague2
    {
      set{ colleague2 = value; }
    }

    public override void Send(string message,
      Colleague colleague)
    {
      if (colleague == colleague1)
      {
```

```
}
// "ConcreteColleague1"
class ConcreteColleague1 : Colleague
{
  // Constructor
  public ConcreteColleague1(Mediator mediator)
    : base(mediator)
  {
  }

  public void Send(string message)
  {
    mediator.Send(message, this);
  }

  // "ConcreteColleague2"
  class ConcreteColleague2 : Colleague
  {
    // Constructor
    public ConcreteColleague2(Mediator med
      : base(mediator)
    {
    }

    public void Send(string message)
    {
      mediator.Send(message, this);
    }
```

```
    message);
  }
}
```

```
s += au;
```

monads are like **design patterns**

When trying to understand them,
you can't point at a specific thing.

Confusing things about monads

- monadic **return** has nothing to do with the imperative **return**
- Why is it called **return** then?

Haskell History

- Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler: **A History of Haskell: Being Lazy With Class**

6.4 Higher-kinded polymorphism

The first major, unanticipated development in the type-class story came when Mark Jones, then at Yale, suggested parameterising a class over a type *constructor* instead of over a *type*, an idea he called *constructor classes* (Jones, 1993). The most immediate and persuasive application of this idea was to monads (discussed in Section 7), thus:

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

Haskell History

- Mark P. Jones: **A system of constructor classes: overloading and implicit higher-order polymorphism**

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ m \Rightarrow \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{result} \quad :: \quad a \to m\ a \\
\quad \mathit{bind} \quad :: \quad m\ a \to (a \to m\ b) \to m\ b \\
\quad \mathit{join} \quad :: \quad m\ (m\ a) \to m\ a \\
\\
\quad x\ \text{`}\mathit{bind}\text{`}\ f \quad = \quad \mathit{join}\ (\mathit{map}\ f\ x) \\
\quad \mathit{join}\ x \quad = \quad x\ \text{`}\mathit{bind}\text{`}\ \mathit{id}
\end{array}
$$

Haskell History

- Philip Wadler: **The essence of functional programming**

2.1 What is a monad?

For our purposes, a *monad* is a triple (`M`, `unitM`, `bindM`) consisting of a type constructor `M` and a pair of polymorphic functions.

```
unitM  ::  a -> M a
bindM  ::  M a -> (a -> M b) -> M b
```

These two functions satisfy three laws, which are discussed in Section 2.10.

Confusing things about monads

- "the monad is a box"
- Someone coming from Java might think that monads are like boxed values. They aren't.

Confusing things about **IO**

- The name of IO
- IO is not just for I/O
- You can't get out of IO,* only the Haskell runtime can.
- Example about separating IO and logic later.

* you can with unsafePerformIO

Confusing things about **State**

- The name of the State monad.
- Reading the type definition for the State monad.

```
newtype State s a =
  State { runState :: s -> (a, s) }
```

```haskell
newtype State s a =
  State { runState :: s -> (a, s) }

return :: a -> State s a
return x = State ( \st -> (x, st) )

(>>=) :: State s a -> (a -> State s b) -> State s b
proc >>= procGen = State $ \st ->
      let (x, st') = runState proc st
      in runState (procGen x) st'
```

State works like the memory buttons on a calculator.
You have a single value you can read, write or modify with a function.

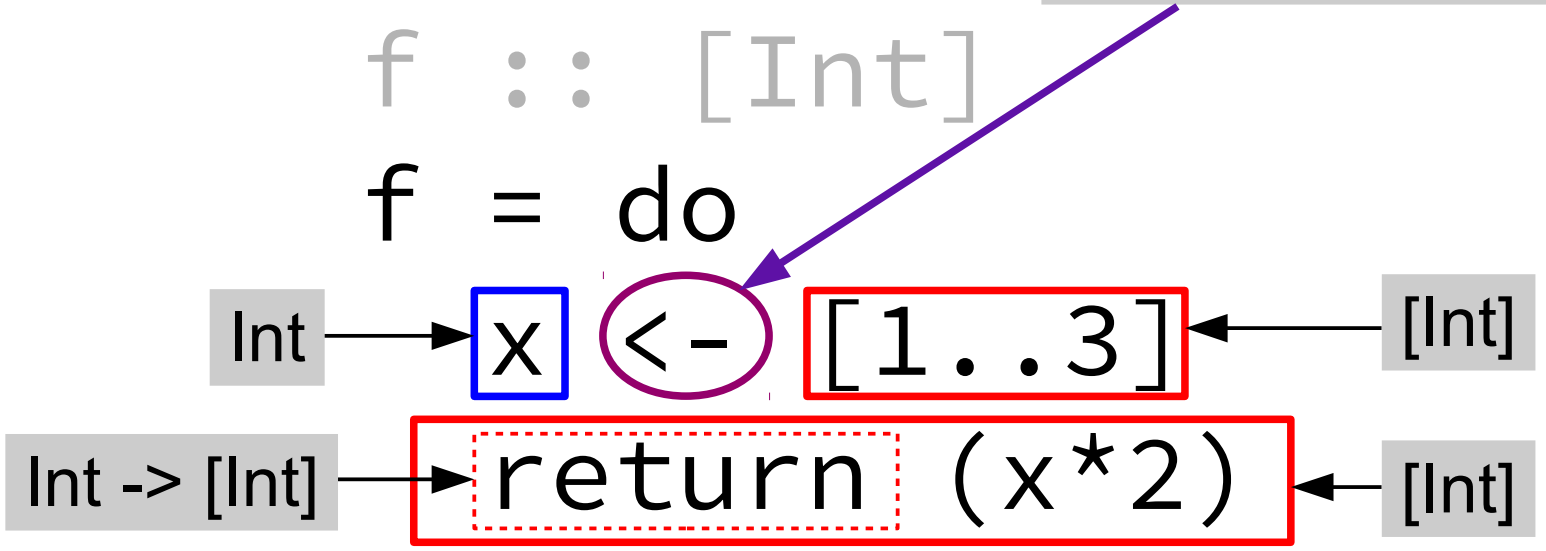
Summary

- Don't read definitions (first)!
- Read examples!
- Names are confusing!
- IO is evil!
- Be careful with State!

Examples

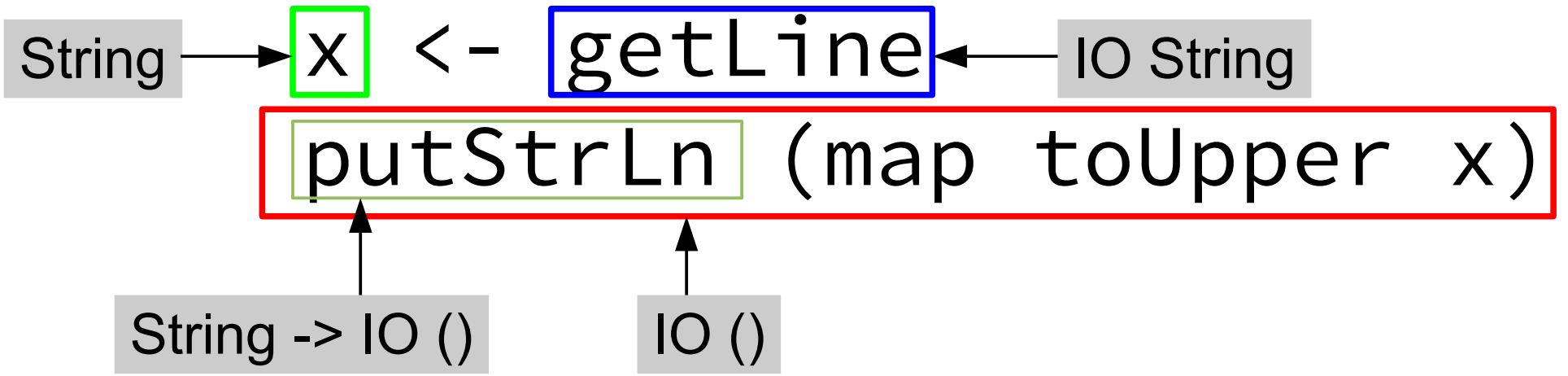
- Three important kinds of examples:
  - Syntactical
  - Functional
  - Structural

"Unpacks" and binds a monadic value to an identifier, **within a do block**.
The type of the **do block** and the type of the **monadic value** must match up!




```
f == [2,4,6]
```

```
g :: IO ()
g = do
  x <- getLine
  putStrLn (map toUpper x)
```




```
>Klaatu Barada Nikto
"KLAATU BARADA NIKTO"
```

```
g :: IO ()
g = do
  x <- getLine
  putStrLn (map toUpper x)
```

COULD WE WRITE THIS DIFFERENTLY?

LET'S TRY!

negative example 1

BAD CODE!!

```
g :: IO ()
g = do
  x <- (map toUpper) getLine
  putStrLn x
```

negative example 1

BAD CODE!!

```
g :: IO ()
g = do
  x <- (map toUpper) getLine
  putStrLn x
```

[Char] -> [Char]

IO [Char]

[Char] is not **IO [Char]** !!

negative example 1

FIX

```
g :: IO ()
g = do
  x <- liftM (map toUpper) getLine
  putStrLn x
```

```
liftM :: Monad m => (a -> r) -> m a -> m r

([Char] -> [Char]) -> IO [Char] -> IO [Char]
```

negative example 2

BAD CODE!!

```
g :: IO ()
g = do
   x <- getLine
   x <- (map toUpper) x
   putStrLn x
```

negative example 2

BAD CODE!!

```
g :: IO ()
g = do
   x <- getLine
   x <- (map toUpper) x
   putStrLn x
```

We're in an **IO block**.

Not an **IO** value, **can't "unpack"** it.

negative example 2

FIX

```
g :: IO ()
g = do
  x <- getLine
  x <- return ((map toUpper) x)
  putStrLn x
```

```
return :: Monad m => a -> m a

[Char] -> IO [Char]
```

```haskell
import Data.List

h :: [[String]] -> String -> [(Int, Int)]
h xs y = do
  (i,ys) <- zip [0..] xs
  Just j <- return (elemIndex y ys)
  return (i,j)

strs =  [ ["ab", "xx"]
        , ["yy", "yy"]
        , ["zz", "zz", "ab"]
        ]

-- h strs "ab" == [(0,0),(2,2)]
```

State

```
insertionSort :: [Int] -> [Int]
insertionSort ls = fst $ execState (
  forM_ [1..(length ls)-1] (\i-> do
    modify $ \(ls,j) -> (ls,i)
    while (\(ls,j) -> j>0 && ls!!(j-1) > ls!!j) $
      modify $ \(ls,j) -> (swap (j-1) j ls, j-1)
    )) (ls, 0)

while cond = whileM_ (gets cond)
```

ST

```haskell
insertionSortST :: (Ord a, Storable a) =>
  MVector s a -> ST s ()
insertionSortST v = do
  j <- newSTRef 0
  CM.forM_ [1..(VSM.length v)-1] (\i-> do
    modifySTRef j $ const i
    whileM_ (cond j)  $ do
      jval <- readSTRef j
      swap v (jval-1) jval
      modifySTRef j (+(-1))
    )
  where
    cond j = do
      jval <- readSTRef j
      if jval>0 then
        do
          x1 <- VSM.read v $ jval-1
          x2 <- VSM.read v jval
          return $ x1 > x2
        else return False
```

```haskell
sort :: Vector Int -> Vector Int
sort vec = runST $ do
  v <- thaw vec
  insertionSortST v
  freeze v
```

JuicyPixels

```
writePixel ::
     PrimMonad m =>
     MutableImage (PrimState m) a
  -> Int
  -> Int
  -> a
  -> m ()
```

JuicyPixels

```
data MutableImage s a = MutableImage
  { mutableImageWidth  :: !Int
  , mutableImageHeight :: !Int
  , mutableImageData   ::
      M.STVector s (PixelBaseComponent a)
  }
```

JuicyPixels

```
type PixelFun = Image PixelRGB8 -> Int -> Int -> PixelRGB8
type ImageFun = Image PixelRGB8 -> Image PixelRGB8

mapImage :: PixelFun -> Image PixelRGB8 -> Image PixelRGB8
mapImage f img@(Image w h idat) = ST.runST $ do
  idat' <- new $ VS.length idat
  let img' = MutableImage w h idat'
  forM_ [(x,y)| x <- [0..w-1], y <- [0..h-1]]
    (\(x, y) -> writePixel img' x y $ f img x y)
  freezeImage img'

compFun :: [PixelFun] -> ImageFun
compFun ls = foldl (.) (mapImage id_fun) $ map mapImage ls
```

Separating IO and pure functions

```
class REPLState s where
  setInput :: String -> s -> s
  getOutput :: s -> String
  running :: s -> Bool

repl :: (REPLState s) => (s -> s) -> StateT s IO ()
repl f = do
  i <- liftIO getLine
  modify $ f.setInput i
  st <- get
  liftIO $ putStrLn $ getOutput st
  when (running st) $ repl f

runRepl :: (REPLState s) => (s -> s) -> s -> IO ()
runRepl f s = evalStateT (repl f) s
```

I hope you enjoyed this presentation!

Thanks for listening!

Thanks to Gábor Páli for helping with this presentation.