Class 2: Algebraic Data Types

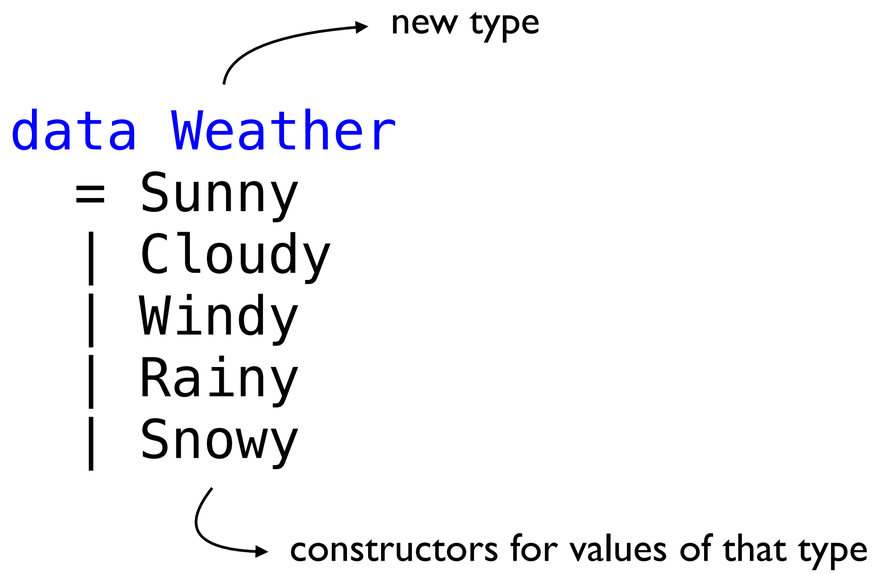
January 24

(reading poll)

(a note on programming idiomatically)

data types by example

note: this weather example is closely sourced from the video
"Algebraic Data Types (ADT) in Scala | Rock the JVM"



enumeration types

(deriving example)

```
w :: Weather
w = Sunny

ws :: [Weather]
ws = [Sunny, Rainy, Snowy]
```

enumeration types

```
isGray :: Weather -> Bool
isGray Sunny  = False
isGray Cloudy = True
isGray Windy  = False
isGray Rainy  = True
isGray Snowy  = False
```

```
isGray :: Weather -> Bool
isGray Cloudy = True
isGray Rainy  = True
isGray w      = False
```

```
isGray :: Weather -> Bool
isGray Cloudy = True
isGray Rainy  = True
isGray _      = False
```

enumeration types

```
data Bool = False | True
```

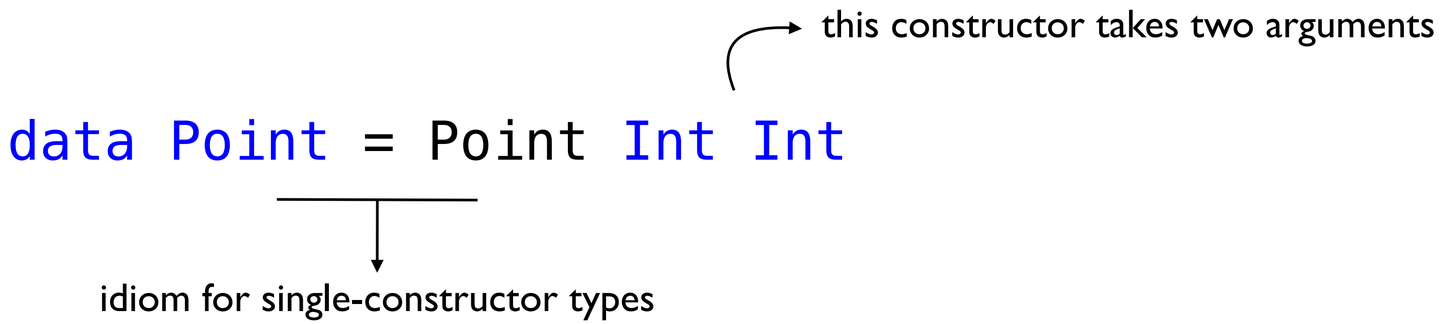
enumeration types

```
or :: Bool -> Bool -> Bool
or True  True  = True
or True  False = True
or False True  = True
or False False = False
```

(boolean polls)

```
or :: Bool -> Bool -> Bool
or True  _  = True
or False b  = b
```

```
(||) :: Bool -> Bool -> Bool
True  || _  = True
False || b  = b
```


this constructor takes two arguments
data Point = Point Int Int
idiom for single-constructor types

```haskell
p :: Point
p = Point 1 2

reflect :: Point -> Point
reflect (Point x y) = Point y x
```

enumeration types

(point poll)

```
data WeatherRequest
  = ByLocation String
  | ByCoordinate Point


data WeatherResult
  = Valid Weather
  | Invalid
```

more data types

(getWeather example)

important: see the reading for sections
on “Pattern Matching” and “Case Expressions”

interlude

```haskell
data Bool = False | True

data Color = Red | Green | Blue
```

```
data Mix = Mix Bool Color Color
```

```
           2  x  3  x  3
```

product types

```
data Unit = Unit
          | I
```

product types

```
data Mix

  = Black                    |
                             +
  | Single Color             3
                             +
  | Double Color Color       9
                             +
  | White                    |
```

sum types

```
data Empty   0
```

sum types

```haskell
data Shape
  = Rectangle Int Int
  | Circle Int
```

```haskell
area :: Shape -> Double
area (Rectangle x y) = …
area (Circle r) = …
```

```java
interface Shape
  double area();
```

```java
class Rectangle implements Shape
  int x, y;
  double area() { … }
```

```java
class Circle implements Shape
  int r;
  double area() { … }
```

readiness for change

data types by example

```
data IntList
  = Nil
  | Cons Int IntList
```

data types can defined in terms of themselves!

```
prod :: IntList -> Int
prod Nil = 1
prod (Cons x xs) = x * prod xs
```

```
data Tree
  = Leaf
  | Node Tree Int Tree
```

recursive types

```
tree :: Tree
tree = Node Leaf 1 (Node Leaf 2 Leaf)
```

(tree exercises)