Recap from week 1: Data types

# Types and constructors

```
data Suit = Spades | Hearts | Diamonds | Clubs
```

Interpretation:

“Here is a new type Suit. This type has four possible values: Spades, Hearts, Diamonds and Clubs.”

# Types and constructors

```
data Suit = Spades | Hearts | Diamonds | Clubs
```

This definition introduces five things:

– The type Suit

– The constructors

```
Spades    :: Suit
Hearts    :: Suit
Diamonds  :: Suit
Clubs     :: Suit
```

# Types and constructors

```
data Rank = Numeric Integer | Jack | Queen | King | Ace
```

Interpretation:

“Here is a new type Rank. Values of this type have five possible possible forms: Numeric n, Jack, Queen, King or Ace, where n is a value of type Integer”

# Types and constructors

```
data Rank = Numeric Integer | Jack | Queen | King | Ace
```

This definition introduces six things:

- The type Rank
- The constructors

```
Numeric    :: ???
Jack       :: ???
Queen      :: ???
King       :: ???
Ace        :: ???
```

# Types and constructors

```
data Rank = Numeric Integer | Jack | Queen | King | Ace
```

This definition introduces six things:

- The type Rank
- The constructors

```
Numeric   :: Integer → Rank
Jack      :: ???
Queen     :: ???
King      :: ???
Ace       :: ???
```

# Types and constructors

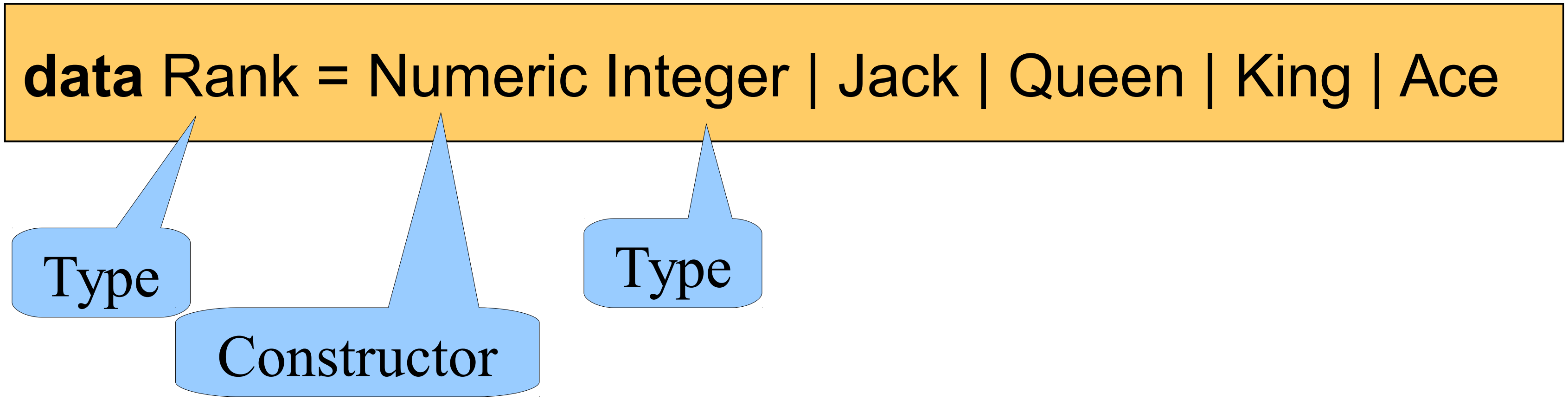
```
data Rank = Numeric Integer | Jack | Queen | King | Ace
```

This definition introduces six things:

- The type Rank
- The constructors

```
Numeric    :: Integer → Rank
Jack       :: Rank
Queen      :: Rank
King       :: Rank
Ace        :: Rank
```

# Types and constructors



```
data Rank = Numeric Integer | Jack | Queen | King | Ace
```

# Types and constructors

```
data Card = Card Rank Suit
```

Interpretation:

“Here is a new type Card. Values of this type have the form Card r s, where r and s are values of type Rank and Suit respectively.”

# Types and constructors

```
data Card = Card Rank Suit
```

This definition introduces two things:

- The type Card
- The constructor

```
Card :: ???
```

# Types and constructors

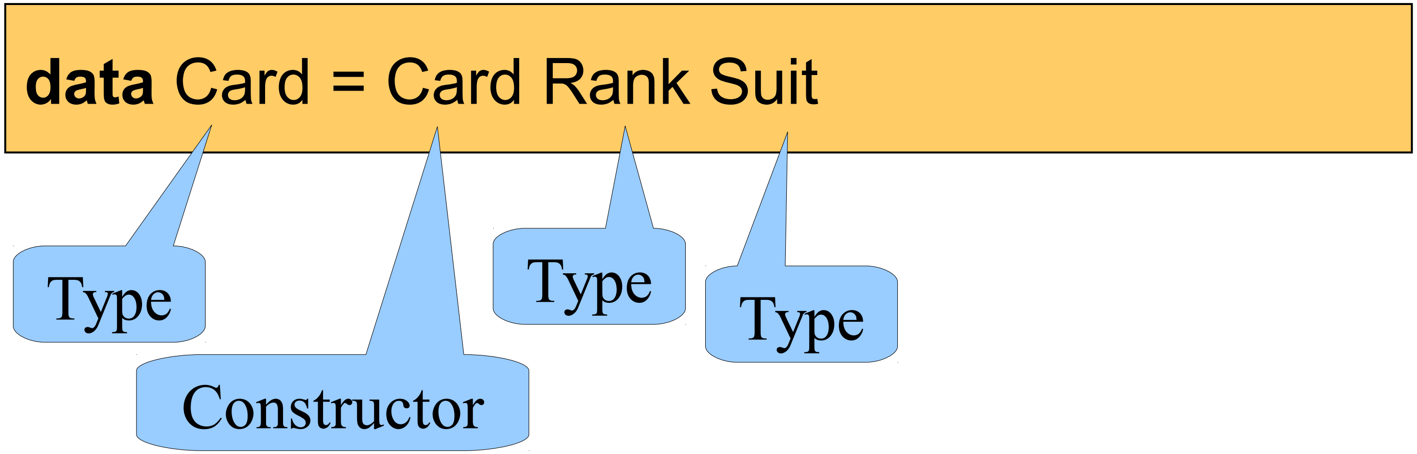
```
data Card = Card Rank Suit
```

This definition introduces two things:

- The type Card
- The constructor

```
Card :: Rank → Suit → Card
```

# Types and constructors



```
data Card = Card Rank Suit
```

# Built-in lists

```
data [a] = []  |  (:) a [a]
```

Not a legal definition, but the built-in lists are *conceptually* defined like this

Constructors:

```
[]      :: [a]
(:)     :: a → [a] → [a]
```

# Some list operations

- From the Data.List module (also in the Prelude):

```
reverse         :: [a] -> [a]
  -- reverse a list

take            :: Int -> [a] -> [a]
  -- (take n) picks the first n elements

(++)            :: [a] -> [a] -> [a]
  -- append a list after another

replicate       :: Int -> a -> [a]
  -- make a list by replicating an element
```

# Some list operations

```
*Main> reverse [1,2,3]
[3,2,1]

*Main> take 4 [1..10]
[1,2,3,4]

*Main> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]

*Main> replicate 5 2
[2,2,2,2,2]
```

# Strings are lists of characters

```
type String = [Char]

Prelude> 'g' : "apa"
"gapa"

Prelude> "flyg" ++ "plan"
"flygplan"

Prelude> ['A','p','a']
"Apa"
```

Type synonym definition

# More on Types

- Functions can have "general" types:
  - *polymorphism*
  - reverse :: [a] → [a]
  - (:)        :: a → [a] → [a]
- Sometimes, these types can be restricted
  - Ord a => … for comparisons (<, <=, >, >=, …)
  - Eq a => … for equality (==, /=)
  - Num a => … for numeric operations (+, -, *, …)

# Do’s and Don’ts

```
isBig :: Integer → Bool
isBig n | n > 9999  = True
        | otherwise = False
```

guards and boolean results

```
isBig :: Integer → Bool
isBig n = n > 9999
```

# Do’s and Don’ts

```
resultIsSmall :: Integer → Bool
resultIsSmall n = isSmall (f n) == True
```

comparison with a boolean constant

```
resultIsSmall :: Integer → Bool
resultIsSmall n = isSmall (f n)
```

# Do’s and Don’ts

```
resultIsBig :: Integer → Bool
resultIsBig n = isSmall (f n) == False
```

comparison with a boolean constant

```
resultIsBig :: Integer → Bool
resultIsBig n = not (isSmall (f n))
```

# nd Don’ts

Do not make unnecessary case distinctions

```
fun1 :: [Integer] → Bool
fun1 []       = False
fun1 (x:xs) = length (x:xs) == 10
```

necessary case distinction?

repeated code

```
fun1 :: [Integer] → Bool
fun1 xs = length xs == 10
```

# s and Don'ts

Make the base case as simple as possible

```
n2 :: [   ger] → Integer
fun2 [x]      = calc x
fun2 (x:xs) = calc x + fun2 xs
```

right base case ?

repeated code

```
fun2 :: [Integer] → Integer
fun2 []       = 0
fun2 (x:xs) = calc x + fun2 xs
```