# Data Types

**Note**: You can ignore the lines after each data declaration that say `deriving ...` – these are there to make it easier for you to play with the examples. If you're curious, though, those lines have to do with type classes.

## Simple Data Declaration

Basic data type declaration:

```
data Bool' = True' | False'
    deriving (Show, Eq)
```

- `Bool'` is the name of our data type
- `True'` and `False'` are the *data constructors*

*Note: `Bool` is already defined in Haskell, which is why the code uses `Bool'`, `True'`, and `False'` here.*

Then we can use the constructors to create instances of this type:

```
true :: Bool'
true = True'

false :: Bool'
false = False'
```

Let's write a few of the functions that are common for booleans:

1. `not`
2. `and`, `or`
3. `xor`

### `not`

```
not :: Bool' -> Bool'
not True'  = False'
not False' = True'
```

### `and`

First pass:

```
and :: Bool' -> Bool' -> Bool'
and False' False' = False'
and False' True'  = False'
```

```
and True'  False' = False'
and True'  True'  = True'
```

Better:

```
and :: Bool' -> Bool' -> Bool'
and True'  True'  = True'
and _      _      = False'
```

### `or`

First pass:

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

Better:

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

### `xor`

First pass:

```
xor :: Bool' -> Bool' -> Bool'
xor False' False' = True'
xor False' True'  = False'
xor True'  False' = False'
xor True'  True'  = True'
```

```
xor :: Bool' -> Bool' -> Bool'
xor False' False' = True'
xor True'  True'  = True'
xor _      _    = False'
```

We can reduce this by one more line (of actual logic), but not with pattern matching (assumes we had a `==` equality comparison function for `Bool'`s):

```
xor :: Bool' -> Bool' -> Bool'
xor b1 b2
    | b1 == b2  = True'
    | otherwise = False'
```

Purpose of this example:

- Pattern matching can *only* compare *concrete values of a given input*
- Pattern matching can be very concise, but is limited in its application

## Data Type with Parameters

To add parameters to a data constructor, simply list each of the *types* of the parameters after the constructor name:

```
-- Person has Name and Age
data Person = Person String Int
```

Note that the *data constructor* (RHS) may have the same name as the *data type* (LHS).

`String` and `Int` are vague, though. Let's clarify by declaring a couple of *type synonyms*.

```
data Person = Person Name Age
    deriving Show
type Name = String
type Age = Int
```

Then we can create a `Person`:

```
bigL = Person "Lamont Coleman" 24
```

### Question

What if we wanted to pattern match on a `Person`?

**Answer:** We do the same thing we did with other data types: we pattern match against *concrete values of the correct type*.

```
bob = Person "Bob" 50

trueIfBob50 :: Person -> Bool
trueIfBob50 (Person "Bob" 50) = True
trueIfBob50 (Person _     _ ) = False

trueIfBob :: Person -> Bool
trueIfBob (Person "Bob" _) = True
trueIfBob (Person _     _) = False

trueIfOld :: Person -> Bool
trueIfOld (Person _ age) = age > 24
```

```haskell
trueIfOldBob :: Person -> Bool
trueIfOldBob (Person "Bob" age) = age > 24
trueIfOldBob (Person _     _  ) = False

-- `head` returns first element of string/list
firstInitial :: Person -> Char
firstInitial (Person name _) = head name
```

*Note: Whenever we don't care about a particular input value, we use an _ to say "I don't care what the input value to this parameter is, and I'm not going to use it on the RHS of the function so don't bother giving it a name".*

#### Tuples vs. Data Constructors

Note that we could also use a tuple to make a single data type out of multiple values. For example, we could define a `Point` as a type synonym of a tuple of `Float`s:

```haskell
type Point = (Float, Float)

point :: Point
point = (1.1, 2.4)
```

The data constructor analog would look like:

```haskell
data Point = Point Float Float
    deriving Show

point :: Point
point = Point 1.1 2.4
```

The difference comes down to judgement and specific cases, but here are the main thoughts to keep in your mind:

- A *type synonym* is simply a different name for a type, so the synonym is interchangeable with the original type.
- A good rule of thumb is to make anything in your code that is especially meaningful/important into it's own data type (rather than a synonym).

#### Type Synonym Example

Let's look at a case where a type synonym fits very well. Imagine we had the `Point` data type we defined above, along with these functions:

```haskell
getX :: Point -> Float
getX (Point x _) = x
```

```haskell
getY :: Point -> Float
getY (Point _ y) = y
```

Now let's say we want to change the `Float` arguments in the `Point` constructor to be `Int`s instead:

```haskell
data Point = Point Int Int
```

However, this breaks our **getX** and **getY** functions, because they each try to extract a `Float` from a `Point`, *not* an `Int`.

A type synonym can help us here. Let's define the type `Coordinate` to mean the same thing as `Float`:

```haskell
data Point = Point Coordinate Coordinate

type Coordinate = Float
```

Now our `Point` contructor accepts two `Coordinate`s. Our **getX** and **getY** now look like:

```haskell
getX, getY :: Point -> Coordinate
getX (Point x _) = x
getY (Point _ y) = y
```

If we came back later wanted to change the `Coordinate` type to be `Int` instead of `Float`, we would only have to change a single line:

```haskell
type Coordinate = Int
```

And since the types of the `Point` constructor and the **getX**, **getY** functions refer to `Coordinate`, we don't have to change anything else and everything should still work.

This example was somewhat contrived though, as there very well may be a part of the code somewhere that is dependent on `Float` and changing the meaning of `Coordinate` to `Int` might break that. A very simple case:

```haskell
point = Point 3.4 5.6
```

If we tried to declare this `point` value but `Point` was expecting two `Int` coordinates, then our code wouldn't compile.

## Data Constructor Types

Like most of what we've dealth with in Haskell, data constructors have types.

#### Question

a. What is the type of `True'`?

**Answer:** `Bool'`

(We can verify in `ghci` with `:t True'`.)

b. What is the type of the *constructor* `Person`?

Consider how we created a `Person`:

```
bob = Person "Bob" 50
```

**Answer:** `Person :: Name -> Age -> Person`

This looks a lot like a function!

## Recursive Data Types: Peano Numbers

The *peano numbers* are a way to represent integers using only a zero value and a successor function.

```
data Peano = Zero | Succ Peano
```

We have two data constructors here

- `Zero`, which has no arguments
- `Succ`, which takes a single argument of type `Peano`
  - This makes `Peano` a *recursively-defined data type*

#### Question

Given:

```
zero = Zero
```

a. How do we create a value `one` that is the successor of `Zero`? How about `two`?

**Answer:**

```
one = Succ zero
two = Succ one
```

b. How else could we constructor `two`?

```
two = Succ (Succ zero)
-- or
two = Succ (Succ Zero)
```

### Question

How can we (exhaustively) convert these into the numbers we're used to? Start with the type signature: `peanoToInt :: Peano -> Int`

**Answer:**

```
peanoToInt :: Peano -> Int
peanoToInt Zero = 0
peanoToInt (Succ Zero) = 1
peanoToInt (Succ (Succ Zero)) = 2
-- this could go on awhile...
```

### Increment/decrement functions

**Goal:** Function `increment` that returns the successor of the input `Peano`.

```
increment :: Peano -> Peano
increment p = Succ p
```

**Goal:** Function `decrement` that returns the predecessor of the input `Peano` number. The predecessor of `Zero` is still `Zero` (we have no way to represent negatives).

```
decrement :: Peano -> Peano
decrement (Succ p) = p
decrement Zero     = Zero
```