# Record Syntax

Right, do you want to know a secret? You can't tell anyone I told you this because it is not very functional. This is a feature of Haskell that just makes development easier. It is called *whispers* *record syntax*.

## Motivation

Imagine you had the following type that stored how many times each baker in Bake Off won Star Baker:

```
data StarBakerCount  = StarBakerCount
                         Int
                         Int
                         Int
                         Int
                         Int
                         Int
                         Int
                         Int
                         Int
                         Int
                         Int
                         Int
```

With a dozen bakers do you think you could remember which `Int` refers to which baker? Me neither.

Now we *could* indicate who is who with comments:

```
data StarBakerCount  = StarBakerCount
                         Int -- Dave
                         Int -- Hermine
                         Int -- Laura
                         Int -- Linda
                         Int -- Loriea
                         Int -- Lottie
                         Int -- Makbul
                         Int -- Marc
                         Int -- Mark
                         Int -- Peter
                         Int -- Rowan
                         Int -- Sura
```

This is better, but when I need to increment Peter's count because he is just making Scotland proud, I still have to match on *allllllll* of the bloody `Int`s:

```
starPeter :: StarBakerCount -> StarBakerCount
starPeter (StarBakerCount dave hermine laura linda loriea lottie makbul marc mark peter rowan sur
```

What a faff!

## Creating a Record Type

I only care about Peter, so he is the only one I should have to bother with! Luckily, this is exactly what record syntax allows me to do.

```
data StarBakerCount = StarBakerCount
  { dave    :: Int
  , hermine :: Int
  , laura   :: Int
```

```
  , linda   :: Int
  , loriea  :: Int
  , lottie  :: Int
  , makbul  :: Int
  , marc    :: Int
  , mark    :: Int
  , peter   :: Int
  , rowan   :: Int
  , sura    :: Int
  }
```

Okay, so, here is the deal. This is basically the same as before, BUT this time we have named fields! As you can see above, we still go "Hi Haskell, I am making a custom data type called StarBakerCount with the Constructor `StarBakerCount`" by using the keyword `data` and the custom data type creation pattern of type-name, equals, constructor-name we are familiar with. Then things get switched up a little. To introduce the fact that we are creating a product type with named fields we use curly braces, within which we will list our field names and types.

What this does is introduce an accessor function for each named field, which says "give me a `StarBakerCount` and I will give you an `Int`" i.e. `peter` secretly has type `peter :: StarBakerCount -> Int`, and for any record type field accessor function, the first arg will always be the type of the record, with the rest being whatever is specified as the type of that field.

If you are ever making a product type in Haskell that has more than three components, I highly recommend using record syntax to save yourself a lot of pain. You wouldn't wanna accidentally match on the wrong person and end up giving Paul Star Baker!

### Using a Record Type

Having a type as a record is very handy for writing functions such as our `starPeter`:

```
starPeter :: StarBakerCount -> StarBakerCount
starPeter count = count {peter = peter count + 1}
```

See how we easily updated Peter's value without having to worry about the others? Prepending the record type with a function accessor name will extract that part of the product type. Appending the record type with curly braces enclosing reassignments of one or many of the fields will update only those fields.