Если вы видите что-то необычное, просто сообщите мне.

PART 1: HASKELL'S SIMPLE DATA TYPES

I first learned about Haskell in college. I've considered why I kept up with Haskell after, even when I didn't know about its uses in industry. I realized there were a few key elements that drew me to it.

In a word, Haskell is elegant. For me, this means we can simple concepts in simple terms. In this series, we're going to look at some of these concepts. We'll see that Haskell expresses a lot of ideas in simple terms that other languages express in more complicated terms. In the first part, we'll start by looking at simple data declarations. If you're already familiar with Haskell data declarations, you can move onto part 2, where we'll talk about sum types!

If you've never used Haskell, now is the perfect time to start! For a quick start guide, download our Beginners Checklist. For a more in-depth walkthrough, read our Liftoff Series!

Like some of our other beginner-level series, this series has a companion Github Repository. All the code in these articles lives there so you can follow along and try out some changes if you want! For this article you can look at the Haskell definitions here, or the Java code, or the Python example.

HASKELL DATA DECLARATIONS

To start this series, we'll be comparing a data type with a single constructor across a few different languages. In the next part, we'll look at multi-constructor types. So let's examine a simple type declaration:

```
data SimplePerson = SimplePerson String String String Int String
```

Our declaration is very simple, and fits on one line. There's a single constructor with a few different fields attached to it. We know exactly what the types of those fields are, so we can build the object. The only way we can declare a SimplePerson is to provide all the right information in order.

```
firstPerson :: SimplePerson
firstPerson = SimplePerson "Michael" "Smith" "msmith@gmail.com" 32 "Lawyer"
```

If we provide any less information, we won't have a SimplePerson! We can leave off the last argument. But then the resulting type reflects that we still need that field to complete our item:

```
incomplete :: String -> SimplePerson
incomplete = SimplePerson "Michael" "Smith" "msmith@gmail.com" 32

complete :: SimplePerson
complete = incomplete "Firefighter"
```

Now, our type declaration is admittedly confusing. We don't know what each field means at all when looking at it. And it would be easy to mix things up. But we can fix that in Haskell with record syntax, which assigns a name to each field.

```
data Person = Person
  { personFirstName :: String
  , personLastName :: String
  , personEmail :: String
  , personAge :: Int
  , personOccupation :: String
  }
```

We can use these names as functions to retrieve the specific fields out of the data item later.

```
fullName :: Person -> String
fullName person = personFirstName person ++ " "
  ++ personLastName person
```

With either construction though, we can also use a pattern match to retrieve the relevant information.

```
fullNameSimple :: SimplePerson -> String
fullNameSimple (SimplePerson fn ln _ _ _) = fn ++ " " ++ ln
```

```
fullName' :: Person -> String
fullName' (Person fn ln _ _ _) = fn ++ " " ++ ln
```

And that's the basics of data types in Haskell! Let's take a look at this same type declaration in a couple other languages.

JAVA

If we wanted to express this in the simplest possible Java form, we'd do so like this:

```
public class Person {
  public String firstName;
  public String lastName;
  public String email;
  public int age;
  public String occupation;
}
```

Now, this definition isn't much longer than the Haskell definition. It isn't a very useful definition as written though! We can only initialize it with a default constructor Person(). And then we have to assign all the fields ourselves! So let's fix this with a constructor:

```
public class Person {
    public String firstName;
    public String lastName;
    public String email;
    public int age;
    public String occupation;

    public Person(String fn,
                  String ln,
                  String em,
                  int age,
                  String occ) {
        this.firstName = fn;
        this.lastName = ln;
        this.email = em;
```

```
        this.age = age;
        this.occupation = occ;
    }
}
```

Now we can initialize it in a sensible way. But this still isn't idiomatic Java. Normally, we would have our instance variables declared as private, not public. Then we would expose the ones we wanted via "getter" and "setter" methods. If we do this for all our types, it would bloat the class quite a bit. In general though, you wouldn't have arbitrary setters for all your fields. Here's our code with getters and one setter.

```
public class Person {
    private String firstName;
    private String lastName;
    private String email;
    private int age;
    private String occupation;

    public Person(String fn,
                  String ln,
                  String em,
                  int age,
                  String occ) {
        this.firstName = fn;
        this.lastName = ln;
        this.email = em;
        this.age = age;
        this.occupation = occ;
    }

  public String getFirstName() { return this.firstName; }
  public String getLastName() { return this.lastName; }
  public String getEmail() { return this.email; }
  public int getAge() { return this.age; }
  public String getOccupation() { return this.occupation; }

  public void setOccupation(String occ) { this.occupation = occ; }
}
```

Now we've got code that is both complete and idiomatic Java.

PUBLIC AND PRIVATE

We can see that the lack of a public/private distinction in Haskell saves us a lot of grief in defining our types. Why don't we do this?

In general, we'll declare our data types so that constructors and fields are all visible. After all, data objects should contain data. And this data is usually only useful if we expose it to the outside world. But remember, it's only exposed as read-only, because our objects are immutable! We'd have to construct another object if we want to "mutate" an existing item (IO monad aside).

The other thing to note is we don't consider functions as a part of our data type in the same way Java (or C++) does. A function is a function whether we define it along with our type or not. So we separate them syntactically from our type, which also contributes to conciseness.

Of course, we do have some notion of public and private items in Haskell. Instead of using the type defintion, we handle it with our module definitins. For instance, we might abstract constructors behind other functions. This allows extra features like validation checks. Here's how we can define our person type but hide it's true constructor:

```
module Person (Person, mkPerson) where

-- We do NOT export the `Person` constructor!
--
-- To do that, we would use:
-- module Person (Person(Person)) where
--   OR
-- module Person (Person(..)) where

data Person = Person String String String Int String

mkPerson :: String -> String -> String -> Int -> String
  -> Either String Person
mkPerson = ...
```

Now anyone who uses our code has to use the mkPerson function. This lets us return an error if something is wrong!

PYTHON

As our last example in this article, here's a simple Python version of our data type.

```
class Person(object):

  def __init__(self, fn, ln, em, age, occ):
    self.firstName = fn
    self.lastName = ln
    self.email = em
    self.age = age
    self.occupation = occ
```

This definition is pretty compact. We can add functions to this class, or define them outside and pass the class as another variable. It's not as clean as Haskell, but much shorter than Java.

Now, Python has no notion of private member variables. Conventions exist, like using an underscore in front of "private" variable names. But you can't restrict their usage outside of your file, even through imports! This helps keep the type definition smaller. But it does make Python a little less flexible than other languages.

What Python does have is more flexibility in argument ordering. We can name our arugments as follows, allowing us to change the order we use to initialize our type. Then we can include default arguments (like None).

```
class Person(object):

  def __init__(self, fn=None, ln=None, em=None, age=None, occ=None):
    self.firstName = fn
    self.lastName = ln
    self.email = em
    self.age = age
    self.occupation = occ
```

```
# This person won't have a first name!
myPerson = Person(
             ln="Smith",
             age=25,
             em="msmith@gmail.com",
             occ="Lawyer")
```

This gives more flexibility. We can initialize our object in a lot more different ways. But it's also a bit dangerous. Now we don't necessarily know what fields are null when using our object. This can cause a lot of problems later. We'll explore this theme throughout this series when looking at Python data types and code.

JAVASCRIPT

We'll be making more references to Python throughout this series as we explore its syntax. Most of the observations we make about Python apply equally well to Javascript. In general, Javascript offers us flexibility in constructing objects. For instance, we can even extend objects with new fields once they're created. Javascript even naturalizes the concept of extending objects with functions. (This is possible in Python, but not as idiomatic).

A result of this though is that we have no guarantees about how which of our objects have which fields. We won't know for sure we'll get a good value from calling any given property. Even basic computations in Javascript can give results like NaN or undefined. In Haskell you can end up with undefined, but pretty much only if you assign that value yourself! And in Haskell, we're likely to see an immediate termination of the program if that happens. Javascript might percolate these bad values far up the stack. These can lead to strange computations elsewhere that don't crash our program but give weird output instead!

But the specifics of Javascript can change a lot with the framework you happen to be using. So we won't cite too many code examples in this series. Remember though, most of the observations we make with Python will apply.

CONCLUSION

So after comparing these methods, I much prefer using Haskell's way of defining data. It's clean and quick. We can associate functions with our type or not, and we can make fields private if we want. And that's just in the one-constructor case! We'll see how things get even more hairy for other languages when we add more constructors! Take a look at part 2 to see how things stack up!

If you've never programmed in Haskell, hopefully this series shows you why it's actually not too hard! Read our Liftoff Series or download our Beginners Checklist to get started!

Revision #1
Created 2022-03-11 06:00:01 UTC by gasick
Updated 2022-03-11 17:11:17 UTC by gasick