# Lab 9: Polymorphism in Haskell

This week's lab is all about polymorphism in Haskell. We've covered one kind of polymorphism (generic/parametric) in lecture, and on this lab you'll explore this idea a bit further. Then, you'll explore another form of polymorphism, familiarizing yourself with how this is implemented in Haskell.

## Starter code

- `Lab9.hs`

## Task 1: Fitting some generic functions

We saw at the end of last week's lecture how generically polymorphic functions often have constraints on their implementation imposed by their type signature, as long as we restrict our attention to *total functions*, which are functions that always terminate and do not raise any errors for any inputs of the given parameter types. For example, the *only* total function with type signature `f :: a -> a` is the identity function!

Your first task is to complete the implementations of the given functions under "Task 1" of the starter code. Note that we haven't provided any documentation about the expected behaviour of these functions, only their type signatures. Despite this, there are very few total functions that will allow the program to compile!

## Task 2: One new generic type, `Maybe`

One of the simplest polymorphic types in Haskell is `Maybe`, which has the following definition:

```haskell
data Maybe a = Nothing | Just a
```

This type typically represents the result of a computation that might succeed or fail. A successful computation is represented using the `Just` data constructor, which wraps a single value—of course, this value can be of different types depending on the computation, which is why `Maybe` is polymorphic. A failed computation is represented by using the `Nothing` constructor. While this value is similar to `None` or `null` in other programming languages, there is a key difference: `Nothing` is explicitly of type `Maybe a`, not `a` itself. This means that the only way a function can return `Nothing` is if its return type uses `Maybe`!

For example:

```haskell
safeDiv :: Int -> Int -> Maybe Int
safeDiv x 0 = Nothing   -- Can't divide by 0
safeDiv x y = Just (div x y)
```

Using the same idea, complete the following functions:

1. `safeHead :: [a] -> Maybe a`, which is like `head` but returns `Nothing` rather than raising an error when its input is empty.
2. `safeTail :: [a] -> Maybe [a]`, analogous to `tail`.
3. `onlyWhen :: (a -> Bool) -> a -> Maybe a`, which succeeds only when its second argument satisfies its first.
4. `try :: (a -> b) -> Maybe a -> Maybe b`. There are only two possible total functions with this type signature. Pick the one that *isn't* a constant function. Watch the brackets when doing pattern-matching!

Note: often, simply using `Nothing` is not satisfactory, as we want to know not just that a computation failed, but *why* it failed. We'll discuss a generalization of this approach in lecture.

## Task 3: Introduction to typeclasses

The other major type of polymorphism we'll study in this course is *ad hoc polymorphism*, which allows the same function (or value) to be given different implementations for different types. If this sounds reminiscent of inheritance in object-oriented programming, it should! Indeed, Haskell supports ad hoc polymorphism through **typeclasses**, which are similar to the concept of *abstract interfaces* in OOP.

Read through the following sections in *Learn You a Haskell*:

- Typeclasses 101, up to and including its description of the `Eq` type class.
- Typeclasses 102 up to and including making the `TrafficLight` type an instance of the `Eq` type class.

Then just to check your understanding, make the given `Shape` data type an instance of the `Eq` type class, according to the following specifications:

1. Two circles are equal if and only if they have the same radius.
2. Two rectangles are equal if and only if their widths are equal and their heights are equal.
3. Two squares are equal if and only if their side lengths are equal.
4. A square is equal to a rectangle if and only if the rectangle's width and height both equal the square's side length.
5. No other pair of shapes are considered equal.