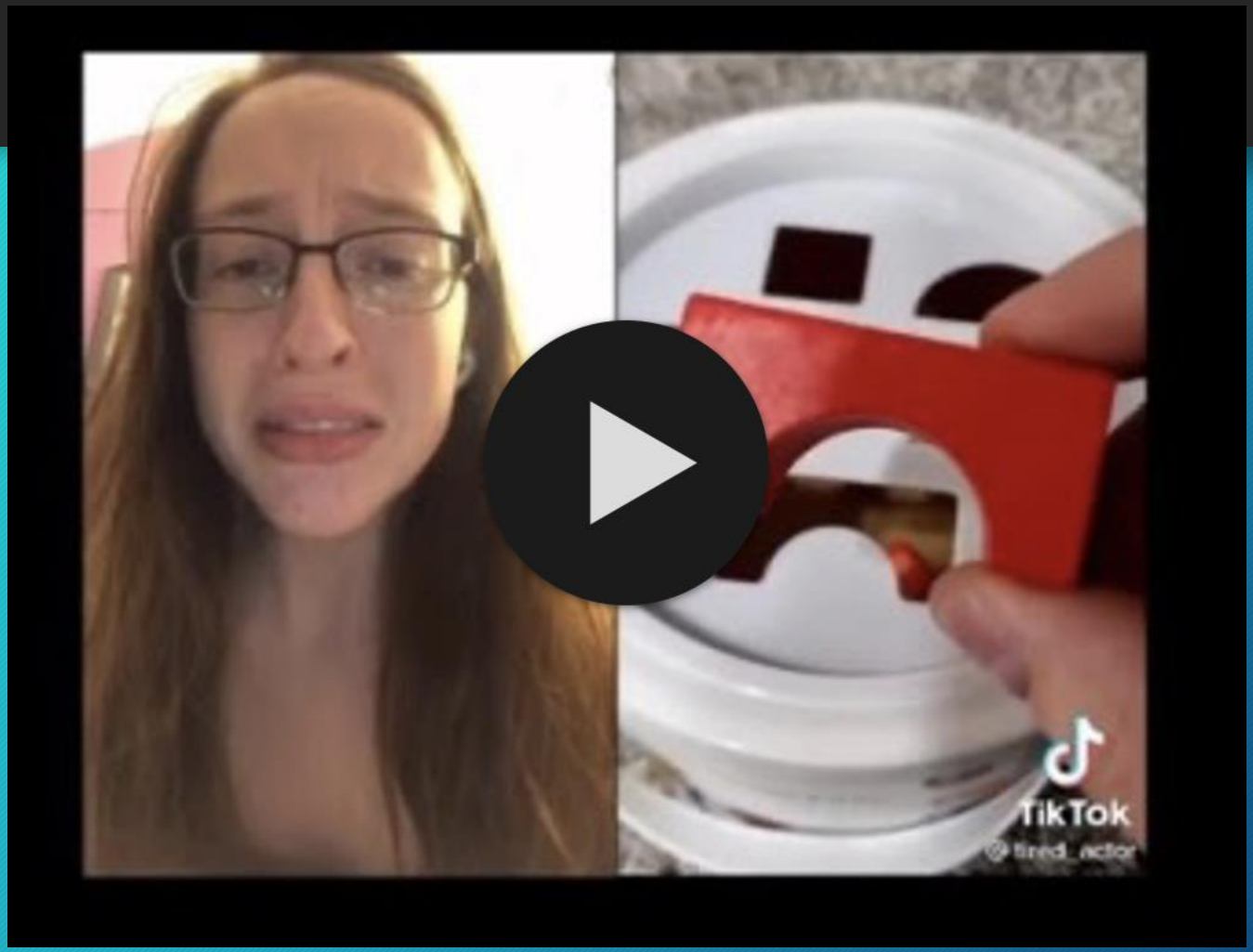
PROGRAMMING IN HASKELL

Topic 3 - Types and Classes

What types should I use?



(https://youtu.be/9nSQs0Gr9FA)

What is a Type?

A type is a name for a collection of related values. For example, in Haskell the basic type

```
Bool
```

contains the two logical values:

```
False
```

```
True
```

Type Errors

Applying a function to one or more arguments of the wrong type is called a type error.

```
> 1 + False
error ...
```

1 is a number and False is a logical value, but + requires two numbers.

Types in Haskell

- If evaluating an expression e would produce a value of type t, then e has type t, written

```
e :: t
```

- Every well formed expression has a type, which can be automatically calculated at compile time using a process called type inference.

Types in Haskell

- All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time.
- In GHCi, the command calculates the type of an expression, without evaluating it:

```
> not False
True

> :type not False
not False :: Bool
```

:type (can be shortened to :t)

Basic Types

Haskell has a number of basic types, including:

- `Bool` - logical values
- `Char` - single characters
- `String` - strings of characters
- `Int` - fixed-precision integers
- `Integer` - arbitrary-precision integers
- `Float` - floating-point numbers

List Types

A list is sequence of values of the same type:

```
[False,True,False] :: [Bool]

['a','b','c','d'] :: [Char]
```

In general:

[t] is the type of lists with elements of type t.

List Types

Note:

- The type of a list says nothing about its length:

```
[False,True] :: [Bool]

[False,True,False] :: [Bool]
```

- The type of the elements is unrestricted. For example, we can have lists of lists:

```
[['a'],['b','c']] :: [[Char]]
```

Tuple Types

A tuple is a sequence of values of different types:

```
(False,True) :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)
```

In general:

(t1,t2,…,tn) is the type of n-tuples whose ith components have type ti for any i in 1…n.

Tuple Types

Note:

- The type of a tuple encodes its size:

```
(False,True) :: (Bool,Bool)

(False,True,False) :: (Bool,Bool,Bool)
```

- The type of the components is unrestricted:

```
('a',(False,'b')) :: (Char,(Bool,Char))

(True,['a','b']) :: (Bool,[Char])
```

Function Types

A function is a mapping from values of one type to values of another type:

```
not :: Bool → Bool

even :: Int → Bool
```

In general:

t1 → t2 is the type of functions that map values of type t1 to values to type t2.

Function Types

Note:

- The arrow → is typed at the keyboard as ->.

- The argument and result types are unrestricted. For example, functions with multiple arguments or results are possible using lists or tuples:

```
add :: (Int,Int) → Int
add (x,y) = x+y

zeroto :: Int → [Int]
zeroto n = [0..n]
```

Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

```
add' :: Int → (Int → Int)
add' x y = x+y
```

add' takes an integer x and returns a function add' x. In turn, this function takes an integer y and returns the result x+y.

Curried Functions

Note:

- add and add' produce the same final result, but add takes its two arguments at the same time, whereas add' takes them one at a time:

```
add :: (Int,Int) → Int

add' :: Int → (Int → Int)
```

- Functions that take their arguments one at a time are called curried functions, celebrating the work of Haskell Curry on such functions.

Curried Functions

- Functions with more than two arguments can be curried by returning nested functions:

```
mult :: Int → (Int → (Int → Int))
mult x y z = x*y*z
```

mult takes an integer x and returns a function mult x, which in turn takes an integer y and returns a function mult x y, which finally takes an integer z and returns the result x*y*z.

Curried Functions

- Functions with more than two arguments can be curried by returning nested functions:

```
mult :: Int → (Int → (Int → Int))
mult x y z = x*y*z
```

mult takes an integer x and returns a function mult x, which in turn takes an integer y and returns a function mult x y, which finally takes an integer z and returns the result x*y*z.

Why is Currying Useful?

Curried functions are more flexible than functions on tuples, because useful functions can often be made by partially applying a curried function.

For example:

```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [Int]
```

Currying Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

- The arrow $\rightarrow$ associates to the right.

```
Int → Int → Int → Int
```

Means Int $\rightarrow$ (Int $\rightarrow$ (Int $\rightarrow$ Int))

Currying Conventions

- As a consequence, it is then natural for function application to associate to the left.

```
mult x y z
```

Means ((mult x) y) z

- Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

Polymorphic Functions

A function is called polymorphic ("of many forms") if its type contains one or more type variables.

```
length :: [a] → Int
```

For any type a, length takes a list of values of type a and returns an integer.

Polymorphic Functions

Note:

- Type variables can be instantiated to different types in different circumstances:

```
> length [False,True]
2
```

a = Bool

```
> length [1,2,3,4]
4
```

a = Int

- Type variables must begin with a lower-case letter, and are usually named a, b, c, etc.

Polymorphic Functions

- Many of the functions defined in the standard Prelude are polymorphic. For example:

```
head :: [a] → a

take :: Int → [a] → [a]

id :: a → a
```

Constrained Types

- **Constrained type variables** can be instantiated to any types that satisfy the constraints:

```
> 1 + 2
3

> 1.0 + 2.0
3.0

> 'a' + 'b'
ERROR
```

a = Int

a = Float

Char is not a numeric type

Type Classes

A type class in Haskell is a collection of types that support certain operations or behaviours. Some common type classes are

- `Num` - Numeric types
- `Eq` - Equality types
- `Ord` - Ordered types

Type Classes - Num

The Num type class is the foundational class for numeric types. It provides basic arithmetic operations like addition, subtraction, multiplication, and negation.

methods

(+) (addition)	**abs (absolute value)**
(-) (subtraction)	signum (sign of the number)
(*) (multiplication)	fromInteger (convert an Integer to any Num type)
negate (negation)	

Example: Int, Integer, Float, Double, Complex

Type Classes - Eq

The **Eq** type class is for types that can be compared for equality or inequality.

methods

(==) (equality)	(/=) (inequality)

Example: Int, Bool, Char, Float, String (nearly all types)

Type Classes - Ord

The Ord type class is for types that can be ordered. It extends Eq (so any type in Ord must also implement Eq) and allows comparison operations like less than, greater than, etc.

methods 

(<) (less than)	**(<=) (less than or equal to)**
(>) (greater than)	(>=) (greater than or equal to)
compare (compares two values)	

Example: Int, Char, Float, Double, String (needed for any sorting function)

Class Constraints

A **class constraint** is used to specify that a type variable must belong to a certain type class.

You can think of it as restricting the type that can be passed to the function. This is useful when the function needs to call methods defined in a particular class (like == from Eq).

Why use Class Constrains?

Class constraints allow you to:

1.Ensure the presence of specific operations: You can restrict the types that your function can work with to only those that support the operations you need (like equality comparison or string conversion).

2.Create more general, reusable code: You can write generic functions that can work with many types, but still enforce certain behaviours or operations.

Overloaded Functions

A polymorphic function is called overloaded if its type contains one or more class constraints.

Note the syntax - Num a =>

```
(+) :: Num a ⇒ a -> a -> a
```

For any numeric type a, (+) takes two values of type a and returns a value of type a.

If there is any more than one class constraints use (Num a, Ord b) =>

Hints and Tips

- When defining a new function in Haskell, it is useful to begin by writing down its type;
- Within a script, it is good practice to state the type of every new function defined; Specifically, you need to do this during this module.
- When stating the types of polymorphic functions that use numbers, equality or orderings, take care to include the necessary class constraints.


ANY
QUESTIONS?