Principles of Programming Languages

Types and Type classes in Haskell

What is a Type?

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

```
Bool
```

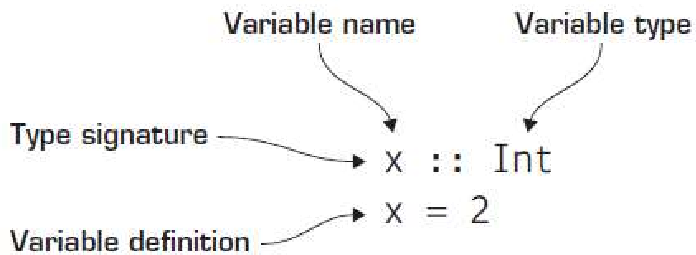
- contains the two logical values:

```
False
```

```
True
```

Types in Haskell

- Haskell uses type inference to automatically determine the types of all values at compile time based on the way they're used.
- All types in Haskell start with a capital letter to distinguish them from functions



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

- A type is: a way to prevent errors
- A type is: a method of organization & documentation
- A type is: a hint to the compiler

A type is a kind of label that every expression has. It tells us in which category of things that expression fits. The expression True is a boolean, "hello" is a string, etc.

Type Inference

Unlike Java or Pascal, Haskell has type inference.

If we write a number, we don't have to tell Haskell it's a number. It can infer that on its own, so we don't have to explicitly write out the types of our functions and expressions to get things done.

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 :type or :t command calculates the type of an expression, without evaluating it:

```
> not False
True

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

Type inference

- What's the difference between type checking and type inference?

```
~~int~~ f(~~int~~ x) {
  return x + 1;
}
```

- Type checking: checks that x is actually used as an int
- Type inference: based usage infers that x is an int

Basic Types

- Haskell has several 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

Int and Integer

- **Int** stands for integer. It's used for whole numbers. 7 can be an Int but 7.2 cannot. Int is bounded, which means that it has a minimum and a maximum value. [*Usually on 32-bit machines the maximum possible Int is 2147483647 and the minimum is -2147483648*].

- **Integer** stands for, er … also integer. The main difference is that it's not bounded so it can be used to represent really big numbers.

Boolean

- The boolean type Bool is an enumeration.
- The basic boolean functions are
  - && (and)
  - || (or)
  - not
- Bool is a boolean type. It can have only two values: True and False.

Characters and Strings

- The character type Char is an enumeration whose values represent Unicode characters.
- Type Char is an instance of the classes Read, Show, Eq, Ord, Enum, and Bounded.
- The toEnum and fromEnum functions, standard functions from class Enum, *map characters to and from the Int type*.
- A string is a list of characters : *type String = [Char]*
- Strings may be abbreviated using the lexical syntax.
- For example, "A string" abbreviates

  ['A',' ','s','t','r', 'i','n','g']

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.

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.

Note:-

- A tuple is fixed in size so we cannot alter it, but List can grow as elements get added.
- The elements of a tuple do not need to be all of the same types, but the list stores only the same type of values.

Note:-

- The type of a tuple encodes its size:

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

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

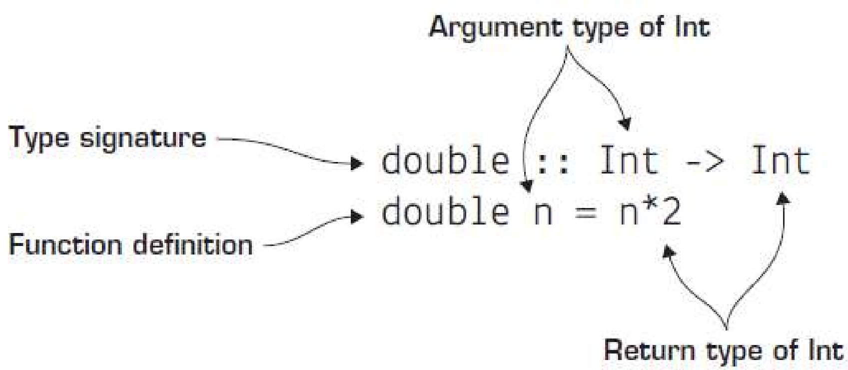
- The type of the components is unrestricted:

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

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

Function Types

- Functions are an abstract type: no constructors directly create functional values.
- Functions also have type signatures. In Haskell an -> is used to separate arguments and return values.



If you define simple by using a type variable, you can imagine that variable being substituted for an actual type when you use that function with various types of arguments.

```
simple :: a -> a
simple x = x
```

```
simple 'a'
```

When you pass a Char to simple, it behaves as though this was its type signature.

```
simple :: Char -> Char
```

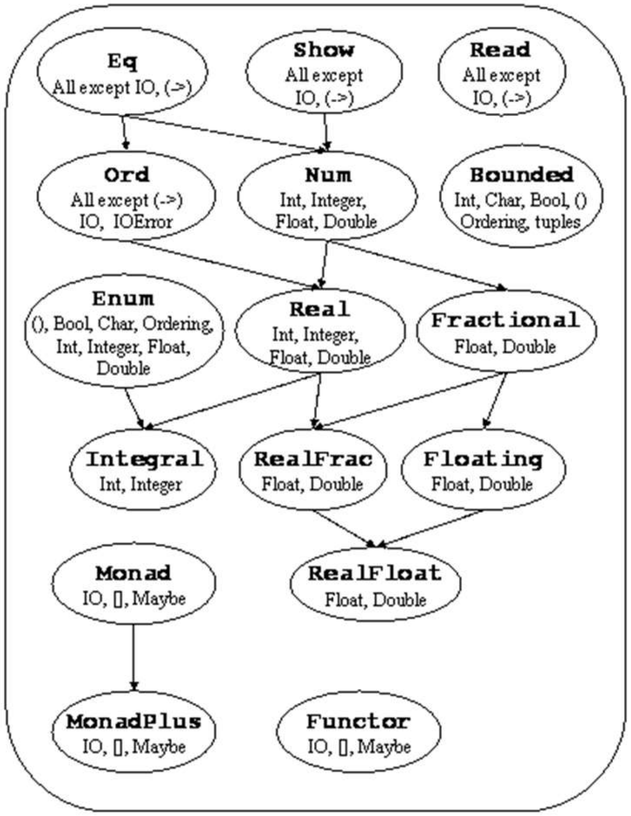
```
simple "dog"
```

Likewise, when you pass a string to simple, it behaves as though it has a different type signature.

```
simple :: String -> String
```

Type classes

Basic Types Classes



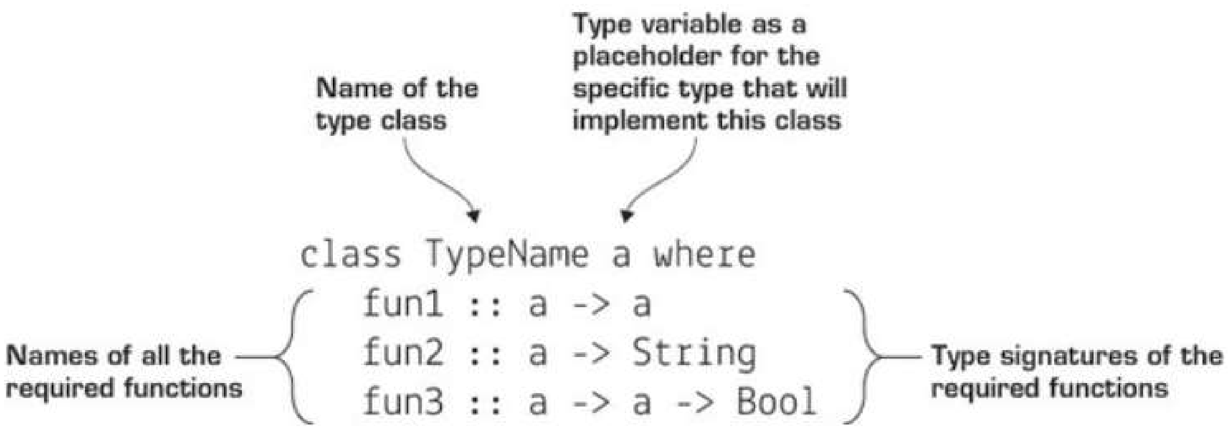
TYPE CLASSES

- Type classes define a set of functions that can have different implementations depending on the type of data they are given.
- Type classes allow you to *group types based on shared behavior*.
- A type class states which function a type must support in the same way that an interface specifies which method a class must support.
- Type classes are the heart of Haskell programming.
- Type classes may remind you of interfaces in Java.

Why Type classes?

- Each function defined works for only one specific set of types.
- Without type classes, you’d need a different name for each function that adds a different type of value.

Structure of a Type class definition



Common Type Classes used

- Eq
- Ord
- Bounded
- Show
- Read
- Num
- Enum

Eq

- The Eq type class needs only two functions: == and /=.
- If you can tell that two types are equal or not equal, that type belongs in the Eq type class.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

```
ghci> 5 == 5
True
ghci> 5 /= 5
False
ghci> 'a' == 'a'
True
ghci> "Ho Ho" == "Ho Ho"
True
ghci> 3.432 == 3.432
True
```

Ord

- Take any two of the same types that implement Ord and return a Boolean.
- Ord covers all the standard comparing functions such as >, <, >= and <=.
- The compare function takes two Ord members of the same type and returns an ordering. Ordering is a type that can be GT, LT or EQ, meaning greater than, lesser than and equal, respectively.

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

To be a member of Ord, a type must first have membership in the prestigious and exclusive Eq club.

```
ghci> "Abrakadabra" < "Zebra"
True
ghci> "Abrakadabra" `compare` "Zebra"
LT
ghci> 5 >= 2
True
ghci> 5 `compare` 3
GT
```

Show

- The show function turns a value into a String.
- All types covered so far except for functions are a part of Show.
- The most used function that deals with the Show typeclass is show. It takes a value whose type is a member of Show and presents it to us as a string.

Every time a value is printed in GHCi, it's printed because it's a member of the Show type class.

```
class Show a where
  show :: a -> String
```

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"
```

Read

- Read is sort of the opposite type class of Show.
- The read function takes a string and returns a type which is a member of Read.

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

- what happens if we try to do just read "4“.

```
ghci> read "4"
<interactive>:1:0:
    Ambiguous type variable `a' in the constraint:
      `Read a' arising from a use of `read' at <interactive>:1:0-7
    Probable fix: add a type signature that fixes these type variable(s)
```

- It knows we want some type that is part of the Read class, it just doesn't know which one. So, use type annotations.

```
ghci> read "5" :: Int
5
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
ghci> read "(3, 'a')" :: (Int, Char)
(3, 'a')
```

Bounded

- Members of Bounded must provide a way to get their upper and lower bounds.
- What's interesting is that minBound and maxBounds aren't functions but values!
- minBound and maxBound are interesting because they have a type of (Bounded a) => a. In a sense they are polymorphic constants.

```
class Bounded a where
  minBound :: a
  maxBound :: a
```

Example:

```
ghci> minBound :: Int
-2147483648
ghci> maxBound :: Char
'\1114111'
ghci> maxBound :: Bool
True
ghci> minBound :: Bool
False
```

Num

- Num is a numeric typeclass. Its members have the property of being able to act like numbers.

```
ghci> :t 20
20 :: (Num t) => t
```

- It appears that whole numbers are also polymorphic constants. They can act like any type that's a member of the Num typeclass.

```
ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
20.0
```

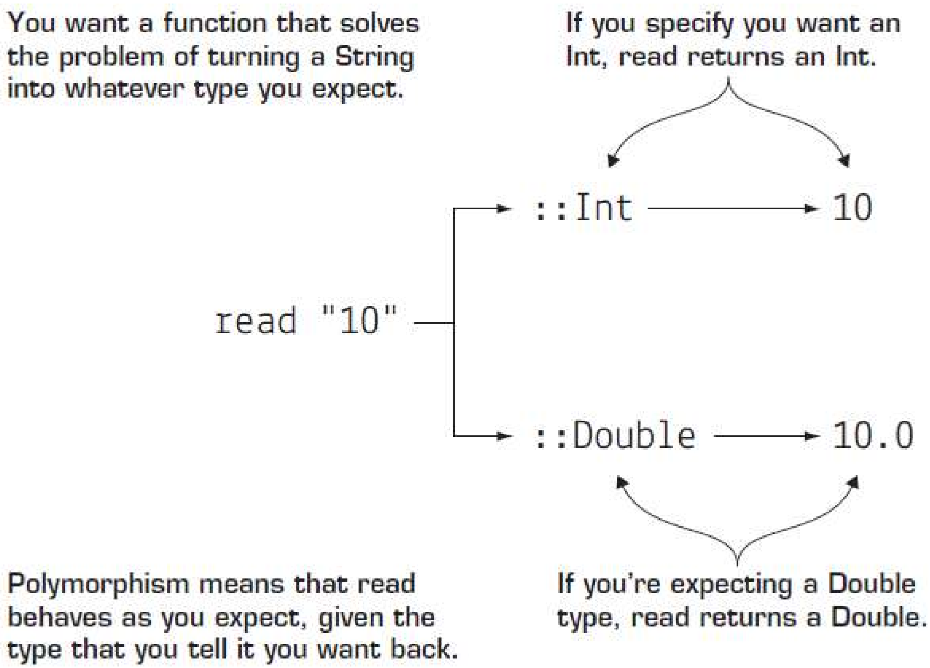
Enum

- Enum members are sequentially ordered types — they can be enumerated.
- The main advantage of the Enum typeclass is that we can use its types in list ranges.
- They also have defined successors and predecesors, which you can get with the succ and pred functions.
- Types in this class: (), Bool, Char, Ordering, Int, Integer, Float and Double.

```
ghci> ['a'..'e']
"abcde"
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> [3 .. 5]
[3,4,5]
ghci> succ 'B'
'C'
```

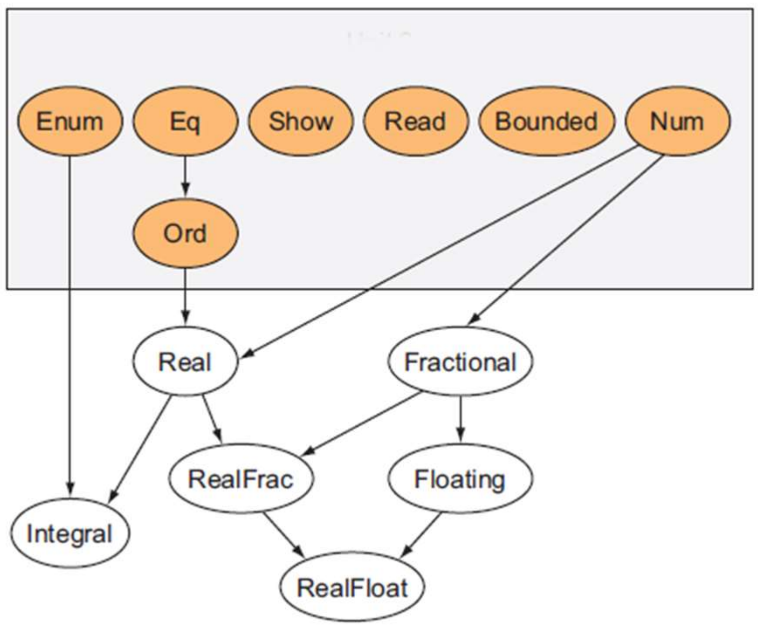
Polymorphism

- Type classes are the way you use polymorphism in Haskell



Type class in Haskell's standard library

Arrows from one class to another indicate a superclass relationship



Next - Functions