The Type of Equality

- We test for equality, using infix operator `==`

  ```
  GHCi> 1 == 2
  False
  GHCi [1,2,3] == (reverse [3,2,1])
  True
  ```

- What is the type of `==` ?
  - It compares things of the same type to give a boolean result:
    `(==) ::  a -> a -> Bool`
    (so it's polymorphic, then?)
  - What does Haskell think ?

    ```
    GHCi> :t (==)
    (==) :: (Eq a) => a -> a -> Bool
    ```

    It says `==` is defined for types that are instances of the the `Eq` Class.

Constraints

- The declaration `Eq a => a -> a -> Bool` contains what is known as a *type constraint* (here, `Eq a =>`)
- The constraint says that the type `a` must belong to the *class of types* `Eq`
- A number of predefined type classes:
  - `Eq` : Defines `==`.
    (Hint: try `:i Eq` in GHCi).
  - `Num` : Defines `+` and `-`, among others
  - `Ord` : Defines comparisons, `<=`
  - `Show` : Can convert to `String`
    (think of implementing `.toString()` in Java).
  - many more...
- The mention of the class name is a promise that some set of functions will work on the values of that class.
- A type class is an *interface* that the compiler will check for you, allowing you to say things like "this function accepts anything that `(+)` works on"

Ad-Hoc Polymorphism

- Equality is "polymorphic"
  `(==) ::  a -> a -> Bool`
- However it is *ad-hoc*:
  - There has to be a specific (different) implementation of it for each type

    ```
    primIntEq :: Int -> Int -> Bool
    primFloatEq :: Float -> Float -> Bool
    ...
    ```

- Contrast with the (parametric) polymorphism of `length`:
  - The same program code works for all lists, regardless of the underlying element type.

    ```
    length [] = 0
    length (x:xs) = 1 + length xs
    ```

Ad-hoc polymorphism is ubiquitous

- Ad-hoc polymorphism is very common in programming languages:

operators	types
$= \neq < \leq > \geq$	$T \times T \to \mathbb{B}$, for (almost) all types T
$+ - * /$	$N \times N \to N$, for numeric types N

- The use of a single symbol ($+$, say) to denote lots of (different but related) operators, is also often called "overloading"
- In many programming languages this overloading is built-in
- In Haskell, it is a language feature called "type classes", so we can "roll our own".

Defining (Type-)Classes in Haskell (Overloading)

- In order to define our own name/operator overloading, we:
  - need to specify the name/operator involved (e.g. `==`);
  - need to describe its pattern of use (e.g. `a -> a -> Bool`;
  - need an overarching "class" name for the concept (e.g. `Eq`).
- In order to use our operator with a given type (e.g. `Bool`, we:
  - need to give the implementation of `==` for that type (`Bool -> Bool -> Bool`).
  - In other words, we define an instance of the type for the class.

Defining The Equality Class

- We define the class `Eq` as follows:

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

- The first line introduces `Eq` as a class characterising a type (here called `a`).
- The second line declares that a type belonging to this class must have an implementation of `==` of the type shown.
- `class` and `where` are Haskell keywords

Giving an instance of the Equality Class

- We define an instance of `Eq` for booleans as follows

```
instance Eq Bool where
  True == True     =  True
  False == False   =  True
  _ == _           =  False
```

  (here `_` is a wildcard pattern matching anything).
- Now all we do is define instances for the other types for which equality is desired.
  - (In fact, in many cases, for equality, we simply refer to a primitive builtin function to do the comparison)
  - Most of this is already done for us as part of the Haskell *Prelude*.
- `instance` is a Haskell keyword

The "real" equality class

- In fact, `Eq` has a slightly more complicated definition:

```
class  Eq a  where
  (==), (/=) :: a -> a -> Bool
     -- Minimal complete definition: (==) or (/=)
  x /= y     =  not (x == y)
  x == y     =  not (x /= y)
```

- First, an instance must also provide `/=` (not-equal).
- Second, we give (circular) definitions of `==` and `/=` in terms of each other
  - The idea is that an instance need only define one of these
  - The other is then automatically derived.
  - However we may want to explicitly define both (for efficiency).

How Haskell handles a class name/operator (I)

- Consider the following (well-typed) expression:
  `x == 3 && y == False`
  (So `x` has type `Int`, and `y` is of type `Bool`).
- The compiler sees the symbol `==`, notes it belongs to the `Eq` class, and then ...
  - seeing `x::Int` deduces (via type inference) that the first `==` has type `Int -> Int -> Bool`
    This is acceptable as it knows of such an instance of `==`
  - Generates code using that instance for that use of equality
  - Does a similar analysis of the second `==` symbol, and generates boolean-equality code there.

How Haskell handles a class name/operator (II)

- Now consider the following (well-typed) expression:
  `x == 3 && y == False || z == MyCons`
  (here `z` has a user defined `data` type `MyType`, with `MyCons` as a constructor).
  Assume we have *not* declared an instance of `Eq` for this type
- The compiler, seeing the 3rd `==`, looks for an instance for `MyType` of `Eq`, and fails to find one
- It generates a error message of the form

```
No instance for (Eq MyType)
  arising from a use of ‘==’ at ...
Possible fix: add an instance declaration for (Eq MyType)
```

- Note the helpful suggestion !

Why not make `Expr` a member of the `Num` Class?

```
instance Num Expr where
  ...
```

This way we could then use standard arithmetic operators like `(+)` and `(*)` directly

```
Val 1.0 + Val 2.0 * Val 3.0
```

So, what does this involve? We need to look at the methods required for the `Num` class.

Guided tour: the `Num a` Class

Class Members

```
(+), (-), (*)     :: a -> a -> a
negate            :: a -> a
abs, signum       :: a -> a
fromInteger       :: Integer -> a
```

Instances	`Int`, `Integer`, `Float`, `Double`
Comments	Required: `Eq`, `Show` Most general notion of number available. (Note lack of any form of division).

Starting the Instance

We shall simply define each class function for `Expr` as a call to an external function, rather than defining them in place.

```
instance Num Expr where
  e1 + e2        =  addExpr e1 e2
  e1 - e2        =  subExpr e1 e2
  e1 * e2        =  mulExpr e1 e2
  negate e       =  negExpr e
  abs e          =  absExpr e
  signum e       =  signumExpr e
  fromInteger i  =  integerToExpr i
```

So far so good, but are storm-clouds looming?
What are the types of functions `addExpr` ... `integerToExpr` ?

Typing the instance functions

We simply replace any occurrence of `a` in the class definition of `Num` by `Expr`.

```
addExpr       ::  Expr -> Expr -> Expr
subExpr       ::  Expr -> Expr -> Expr
mulExpr       ::  Expr -> Expr -> Expr
negExpr       ::  Expr -> Expr
absExpr       ::  Expr -> Expr
signumExpr    ::  Expr -> Expr
integerToExpr ::  Integer -> Expr
```

Ok, let's tackle `addExpr`

Implementing (+) for Expr

Simplest approach, `(+)` for `Expr` is simply `Add`!

```
addExpr e1 e2 = Add e1 e2  -- or addExpr = Add !
```

```
> (Val 1.0) + (Val 1.0)
Add (Val 1.0) (Val 1.0)
```

Hmmm, maybe we'd prefer the following?

```
> (Val 1.0) + (Val 1.0)
(Val 2.0)
```

Implementing (+) for Expr using simp

Lets use `simp` to see how far we can push things

```
addExpr e1 e2 = simp (Add e1 e2)  -- or addExpr = Add !
```

```
> (Val 1.0) + (Val 1.0)
(Val 2.0)
>(Var "x") + (Val 0.0)
(Var "x")
>(Var "x") + (Val 1.0)
Add (Var "x") (Val 1.0)
```

We can't use the Exercise One variant of `simp` that takes a dictionary. Why Not?

What dictionary would we use for `(Var "x") + (Val 1.0)` ?
There is no way to supply one, other than a built-in fixed dictionary behind the scenes.

Moving on with `Expr` as `Num`

- Cases `Sub` and `Mul` are very similar to `Add`
- `negExpr` is trickier, but the following will do:

  ```
  negExpr e = simp (Sub (Val 0.0) e)
  ```

- `integerToExpr` is easy but seems strange:

  ```
  integerToExpr i = Val (fromInteger i)
  ```

  Here, `fromInteger` refers to the instance of `fromInteger` defined for the `Double` instance of `Num`.

`Expr` is not adequate for `Num`

Now we run into trouble:

- `abs` cannot be implemented using any combination of add, subtract, multiply or divide.
  The only way forward would be to define a new `Expr` variant to represent the application of the absolute value operator, *e.g.*: `Abs Expr`
- `signum` cannot be implemented using any combination of add, subtract, multiply or divide.
  Again, the only way forward would be to define a new `Expr` variant to represent the application of the signume operator, *e.g.*: `SigNum Expr`

If this is worth it depends on our plans for `Expr` ...