Reading for next class

- Learn You a Haskell
  - Chapter 8
  - Chapter 11

CS 252:

Advanced Programming Language Principles

Algebraic Data Types, Kinds, & Typeclasses

Prof. Tom Austin

San José State University

What happens when we run this code?

```
public class Maybe {
  public static String reverse(String s) {
    return new StringBuilder(s).reverse();
  }

  public static void main(String[] args) {
    String rev = reverse("Racecar");
    System.out.println(rev);
  }
}
```

Compiler error

```
$ javac Maybe.java
Maybe.java:3: error: incompatible types:
StringBuilder cannot be converted to String
    return new StringBuilder(s).reverse();
                                ^
1 error
```

We needed a String but tried to return a StringBuilder.

What happens with *this* code?

```
public class Maybe {
  public static String reverse(String s) {
    return ""
         + new StringBuilder(s).reverse();
  }

  public static void main(String[] args) {
    String rev = reverse("Racecar");
    System.out.println(rev);
  }
}
```

Success!

```
$ javac Maybe.java
$ java Maybe
racecaR
$
```

The types match, so:

1. the code compiles
2. run-time errors are avoided

Except…

```
public class Maybe {
  public static String reverse(String s) {
    return ""
        + new StringBuilder(s).reverse();
  }

  public static void main(String[] args) {
    String rev = reverse(null);
    System.out.println(rev);
  }
}
```

Run-time error

```
$ javac Maybe.java
$ java Maybe
Exception in thread "main" java.lang.NullPointerException
        at java.lang.StringBuilder.<init>(StringBuilder.java:112)
        at Maybe.reverse(Maybe.java:3)
        at Maybe.main(Maybe.java:8)
```

Types are supposed to prevent run-time errors. Why did they fail here?

Null Pointer Exceptions

- Why does Java allow `null`?
- Can we get the same flexibility in Haskell?
- Can we keep type safety?

The Maybe Type

- The option type
- Used when
  - a function might not return a value
  - a caller might not pass in an argument
- ```
  data Maybe a = Nothing
               | Just a
  ```
  ```

```
divide :: Int -> Int -> Maybe Int
divide x 0 = Nothing
divide x y = Just $ x `div` y

test :: Int -> Int
test d = case 1 `divide` d of
  Just n  -> n
  Nothing -> error "Can't divide by zero"

main = do
  putStrLn $ show $ test 9
  putStrLn $ show $ test 0
```

```
import qualified Data.Map as Map
m = Map.empty
m' = Map.insert "a" 42 m
case (Map.lookup "a" m') of
  Just i  -> putStrLn $ show i
  Nothing -> error "Key not found"
```

`Maybe` is an *algebraic data type (ADT)*

An ADT is a *composite* data type;
a type made up of other types.

Can we create our own ADTs?

A type for trees…

`data` keyword lets us define a new type.

```
data Tree =
    Empty
  | Node Tree Tree String
deriving (Show)
```

This works for trees of Strings, but what if we wanted a tree of Ints?

A tree type using type parameters

```
data Tree k =
       Empty
     | Node (Tree k) (Tree k) k
     deriving (Show)
```

k is a *type parameter*

Types of trees

What is the type of `Tree`? And of `Tree Int`?

Trick question: types don't have types.

So what is the type of Node?

```
*Main> :t Node
Node :: Tree k -> Tree k
                  -> k -> Tree k
```

Higher-order functions review

```
*Main> :t (++)
(++) :: [a] -> [a] -> [a]
```

`++` takes a list of `a`'s and returns…

funct. that takes a list of `a`'s and returns…

a list of `a`'s.

Type of a value constructor

```
*Main> :t Node
Node :: Tree k -> Tree k
     -> k -> Tree k
```

We can partially apply `Node`

A leaf function

```
> leaf = Node Empty Empty
```

Now we can define a tree as:

```
> Node (leaf 3) (leaf 7) 5
```

instead of:

```
> Node (Node Empty Empty 3)
>      (Node Empty Empty 7) 5
```

Kinds

What is the type of `Tree` again?

Trick question.

So what is the *kind* of Tree?

```
*Main> :kind Tree
Tree :: * -> *
```

A *kind* is the "type of a type".

Kinds continued

- Primitive types have a kind of "*"

```
*Main> :k String
String :: *
*Main> :k (Int->String)
[Int] :: *
```

- Types with type parameters have more elaborate kinds:

```
*Main> :k Maybe
Maybe :: * -> *
*Main> :k Map
Map :: * -> * -> *
*Main> :k (Map String)
(Map String) :: * -> *
```

Typeclasses

- Similar to interfaces in Java
  - Like a contract
  - Implementation details can be included
- No relation to classes in object-oriented languages.

Eq typeclass

```
class Eq a where
      (==) :: a -> a -> Bool
      (/=) :: a -> a -> Bool
      x == y = not (x /= y)
      x /= y = not (x == y)
```

Adding Eq functionality to Maybe

```
instance Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

This does not quite work… We don't know that `x` and `y` can be compared with `Eq`.

Adding Eq functionality to Maybe

```
instance (Eq m) => Eq (Maybe m)
  where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

`(Eq m) =>` specifies a *class constraint*. In other words, m must support `Eq` functionality.

Type and kind with constraints

```
Prelude> :t 3
3 :: Num a => a

Prelude> :k Num
Num :: * -> Constraint
```