# Курс: Функциональное программирование

# Лекция 6. Классы типов

**Денис Николаевич Москвин**

28.10.2011

Кафедра математических и информационных технологий
Санкт-Петербургского академического университета

# План лекции

- Классы типов
- Реализация
- Стандартные классы типов

# План лекции

- Классы типов
- Реализация
- Стандартные классы типов

## Параметрический полиморфизм

Рассмотрим функцию

```
id       :: a -> a
id x     =  x
```

Её код универсален, то есть:

- годен для использования с параметром любого типа;
- не зависит ни от какой специфики этого типа.

```
id True     :: Bool
id "Hello"  :: [Char]
id id       :: a -> a
```

## Специальный (ad hoc) полиморфизм (1)

Рассмотрим функцию, определяющую, имеется ли элемент в списке:

```
elem            :: a -> [a] -> Bool
elem _ []        = False
elem x (y:ys)    = x==y || elem x ys
```

Для любых ли типов элементов она подходит?

.

## Специальный (ad hoc) полиморфизм (2)

Рассмотрим функцию, определяющую, имеется ли элемент в списке:

```
elem            ::  a -> [a] -> Bool
elem _ []        = False
elem x (y:ys)    = x == y || elem x ys
```

Для любых ли типов элементов она подходит?

Да, но только если оператор сравнения универсален:

```
(==) :: a -> a -> Bool
```

Хорошо ли это?

## Специальный (ad hoc) полиморфизм (3)

Рассмотрим, например, две разные реализации функции-генератора цифр числа $\pi$

```
getNthPiDigit1 :: Integer -> Digit
getNthPiDigit1 n = ...

getNthPiDigit1 :: Integer -> Digit
getNthPiDigit2 n = ...
```

Можно ли утверждать, что

```
getNthPiDigit1 == getNthPiDigit2
```

# Класс типов `Eq`

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

infix  4  ==, /=
```

Имя класса типов задаёт не тип, а ограничение, называемое *контекстом*:

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

elem             :: Eq a =>  a -> [a] -> Bool
elem _ []        =  False
elem x (y:ys)    =  x == y || elem x ys
```

## Объявления представителей (instance declaration)

Тип `a` является *представителем* класса, если для него реализованы определения операций этого класса:

```
instance  Eq Integer  where
    (==) = eqInteger
    (/=) = neqInteger

instance Eq Char where
    (C# c1) == (C# c2) = c1 `eqChar#` c2
    (C# c1) /= (C# c2) = c1 `neChar#` c2

instance Eq Double where
    (D# x) == (D# y) = x ==## y
```

## Полиморфизм при объявлении представителей

Тип-представитель класса может быть полиморфным

```
instance (Eq a) => Eq [a] where
    []     == []     = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _      == _      = False
```

Контекст (в данном случае `(Eq a) =>`) можно использовать при объявлении представителя. Без него такое определение приведёт к ошибке при проверке типа.

## Методы по умолчанию

Выше мы могли определять перегрузку только `(==)`, поскольку в определении класса типов `Eq` имеется реализация по умолчанию для метода `(/=)`

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

Конечно же методы по умолчанию могут быть перегружены в объявлениях представителя (например, из соображений эффективности).

# Производные представители (Derived Instances)

```
data Point a = Point a a deriving Eq

*Fp06> Point 3 5 == Point 3 2
False
*Fp06> Point 3 5 == Point 3.0 5.0
True
*Fp06> Point 3 5 == Point 'a' 'b'
<interactive>:1:9:
    No instance for (Num Char) ...
```

Задав ключ `-XStandaloneDeriving` в прагме `OPTIONS_GHC` можно использовать отдельностоящие объявления

```
deriving instance Show a => Show (Point a)
```

## Расширение класса (Class Extinsion)

Класс `Ord` наследует все методы класса `Eq` плюс содержит собственные методы

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

-- пример
sort :: (Ord a) => [a] -> [a]
```

Допустимо и множественное наследование

```
class (Eq a, Show a) => MyClass a where
    ...
```

## Типовые операторы в объявлениях класса

Переменная типа, параметризующая класс, может иметь кайнд отличный от `*`

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map

instance  Functor Maybe  where
    fmap _ Nothing       = Nothing
    fmap f (Just a)      = Just (f a)

instance Functor ((->) r) where
    fmap = (.)
```

## Сравнение с другими языками

▶ В ООП-языках классы содержат и данные и методы; в Haskell'е их определения разнесены.

▶ Методы класов в Haskell'е напоминают виртуальные функции в C++.

▶ Классы типов похожи на интерфейсы в Java. Они определяют протокол использывания объекта, а не сам объект.

## План лекции

- Классы типов
- Реализация
- Стандартные классы типов

## Реализация классов типов: словари

Классы типов реализуются через механизм передачи словарей (Dictionaries).
Словарь для `Eq'`, то есть запись из его методов

```
data Eq' a = MkEq (a -> a -> Bool) (a -> a -> Bool)
```

Функции-селекторы выбирают методы равенства и неравенства из этого словаря

```
eq :: Eq' a -> a -> a -> Bool
eq (MkEq e _) = e
ne :: Eq' a -> a -> a -> Bool
ne (MkEq _ n) = n
```

## Реализация объявлений представителей

Объявления представителей транслируются в функции, возвращающие словарь...

```
dEqInt :: Eq' Int
dEqInt =  MkEq eqInt (\x y -> not $ eqInt x y)
```

... или в функции, принимающие некоторый словарь и возвращающие более сложный словарь

```
dEqList :: Eq' a -> Eq' [a]
dEqList (MkEq e _) =  MkEq el (\x y -> not $ el x y)
    where el []     []     = True
          el (x:xs) (y:ys) = x `e` y && xs `el` ys
          el _      _      = False
```

## Использование словаря вместо контекста

Функция `elem` теперь принимает словарь в качестве явного параметра

```
elem'              :: Eq' a -> a -> [a] -> Bool
elem' _ _ []       =  False
elem' d x (y:ys)   =  eq d x y || elem' d x ys
```

```
*Fp06> elem' dEqInt 2 [3,5,2]
True
*Fp06> elem' dEqInt 2 [3,5,7]
False
*Fp06> elem' (dEqList dEqInt) [3,5] [[4],[1,2,3],[3,5]]
True
*Fp06> elem' (dEqList dEqInt) [3,5] [[4],[1,2,3],[3,8]]
False
```

## План лекции

- Классы типов
- Реализация
- Стандартные классы типов

# Класс `Ord`

Минимальное полное определение: `compare` или `<=`.

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

    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT
    x <  y = case compare x y of { LT -> True;  _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >  y = case compare x y of { GT -> True;  _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }
    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
```

# Классы Enum и Bounded

Минимальное полное определение: `toEnum` и `fromEnum`.

```
class  Enum a   where
    succ, pred          :: a -> a
    toEnum              :: Int -> a
    fromEnum            :: a -> Int

    enumFrom            :: a -> [a]             -- [n..]
    enumFromThen        :: a -> a -> [a]        -- [n,n'..]
    enumFromTo          :: a -> a -> [a]        -- [n..m]
    enumFromThenTo      :: a -> a -> a -> [a]   -- [n,n'..m]

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

## Класс `Num`

Минимальное полное определение: все, кроме `negate` и `(-)`.

```
class (Eq a, Show a) => Num a where
    (+), (-), (*) :: a -> a -> a
    negate :: a -> a
    abs, signum :: a -> a
    fromInteger :: Integer -> a

    x - y = x + negate y
    negate x = 0 - x
```

Контекста `Ord` нет — для комплексных, например, он лишний.

## Наследники класса `Num`

У `Num` два субкласса:

- `Integral` — целочисленное деление (через `Real`);
- `Fractional` — обычное деление.

`Integer` и `Int` — представители класса `Integral`.

`Float` и `Double` — наследники `Fractional` через довольно длинную иерархию со множественным наследованием.

Автоматического приведения чисел от одного типа к другому в Haskell'е нет.

# Преобразования от целых и к целым

```
*Fp06> :t fromIntegral
fromIntegral :: (Num b, Integral a) => a -> b
*Fp06> :t sqrt
sqrt :: Floating a => a -> a
*Fp06> sqrt 4
2.0
*Fp06> sqrt (4::Int)
<interactive>:1:1:
    No instance for (Floating Int) ...

*Fp06> sqrt $ fromIntegral (4::Int)
2.0
```

## В обратную сторону

```
ceiling, floor, truncate, round :: (RealFrac a, Integral b) => a -> b
```

# Преобразования к рациональным дробям

```
data  (Integral a) => Ratio a -- = !a :% !a  deriving (Eq)
type  Rational  =  Ratio Integer
(%) :: (Integral a) => a -> a -> Ratio a
numerator, denominator :: (Integral a) => Ratio a -> a

*Fp06> :t toRational
toRational :: Real a => a -> Rational
*Fp06> toRational 2.5
5 % 2
*Fp06> 10 % 5
<interactive>:1:4: Not in scope: `%'

*Fp06> :m +Data.Ratio
*Fp06 Data.Ratio> 10 % 5
2 % 1
*Fp06 Data.Ratio> 1 % 3 + 1 % 6
1 % 2
```

## Преобразования к рациональным дробям (2)

Но числа с плавающей точкой лучше конечно не преобразовывать, а аппроксимировать:

```
*Fp06 Data.Ratio> toRational 4.9
2758454771764429 % 562949953421312
*Fp06 Data.Ratio> approxRational 4.9 0.1
5 % 1
*Fp06 Data.Ratio> approxRational 4.9 0.01
49 % 10
```