

# Classes e Instâncias em Haskell

---

Márcio Lopes Cornélio

Centro de Informática - UFPE

## Funções polimórficas?

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

```
(<) :: t -> t -> Bool
```

```
show :: t -> String
```

Funcionam para **qualquer** tipo de dado?

## Classes

Úteis para permitir sobrecarga (*overloading*) de nomes

**Exemplo** Operação de igualdade ==  
Diferentes significados para diferentes tipos

## Funções monomórficas

Definição para apenas **um tipo** de dado específico

```
capitalize :: Char -> Char  
capitalize ch = chr (ord ch + offset)  
where offset = ord 'A' - ord 'a'
```

## Funções polimórficas

### Polimorfismo

Uma única definição pode ser usada para **diversos tipos** de dados

Casos em que o tipo de dado **não é importante**

Similar a *Generics* e *Templates*

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

4

## Overloading

### Caso geral

```
elemGen :: (a -> a -> Bool) -> a -> [a] -> Bool
```

para qualquer parâmetro de tipo (a -> a -> Bool)

6

## Overloading

```
elemBool :: Bool -> [Bool] -> Bool
elemBool x [] = False
elemBool x (y:ys) =
  (x == Bool y) || elemBool x ys
```

```
elemInt :: Int -> [Int] -> Bool
```

```
elemGen :: (a -> a -> Bool) -> a -> [a] -> Bool
```

5

## Overloading

### Sobrecarga

A função pode ser usada para vários (alguns) tipos de dados

definições distintas para cada tipo, diferentemente de funções polimórficas

### Classe

**Coleção de tipos** para os quais uma função está definida

O conjunto de tipos para os quais (==) está definida é a classe igualdade, **Eq**

7

## Definindo a classe igualdade

Identifica-se o que é necessário para um tipo  $t$  ser da classe

Para a classe de igualdade, deve possuir uma função  $(==)$  definida sobre  $t$ , do tipo  $t \rightarrow t \rightarrow \text{Bool}$

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

8

## Funções que usam igualdade

```
allEqual :: Int -> Int -> Int -> Bool ?
allEqual n m p = (n == m) && (m == p)
```

```
allEqual :: t -> t -> t -> Bool
allEqual n m p = (n == m) && (m == p)
```

```
allEqual succ succ succ ?
```

10

## Instâncias

### Definição

**Tipos** membros de uma classes são chamados de **instância** (da classe)

### Exemplo de instâncias de Eq

São instâncias de **Eq** os tipos primitivos e as listas e tuplas de instâncias de **Eq**

```
Int, Float, Char, Bool, [Int]
(Int, Bool), [[Char]], [(Int, [Bool])]
```

$(\text{Int} \rightarrow \text{Int})$  não é instância da classe Eq

Instância de uma classe vs. Instância de um tipo

9

## Contexto

```
member :: [Char] -> Char -> Bool ?
member [] b = False
member (a:as) b = (a==b) || member as b
```

### Contexto

Definido pela parte antes do operador  $=>$

```
member :: Eq t => [t] -> t -> Bool
member [] b = False
member (a:as) b = (a==b) || member as b
```

11

## Derivando instâncias de classes

```
data List t = Nil | Cons t (List t)
              deriving (Eq, Ord, Show)
data Tree t = NilT |
              Node t (Tree t) (Tree t)
              deriving (Eq, Ord, Show)
```

Aplica-se ao tipo algébrico, não aos seus parâmetros Aplica funções (igualdade, exibição, etc.) item a item (pode não ser o ideal)

12

## Definindo **assinaturas** de classes

Funções (nome e tipo) que devem ser definidas **para cada instância da classe**

### Exemplo de assinatura de classe

```
class Visible t where
  toString :: t -> String
  size :: t -> Int
```

13

## Definindo **instâncias** de uma classe

Definir as funções da assinatura para um tipo

### Exemplo de uma instância da classe **Eq**

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

14

## Exemplo de instâncias da classe **Visible**

```
instance Visible Char where
  toString ch = [ch]
  size _ = 1
```

```
instance Visible Bool where
  toString True  = "True"
  toString False = "False"
  size _ = 1
```

15

## Exemplo de instância da classe **Visible**

```
instance Visible t => Visible [t] where
  toString = concat.(map toString)
  size = (foldr (+) 0).(map size)
```

16

## Definições default

```
class Eq t where
  (==), (/=) :: t -> t -> Bool
  a /= b = not (a==b)
```

Podem ser substituídas (*overridden/overwritten*)

17

## Classes derivadas

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

```
iSort :: Ord t => [t] ->[t]
```

Herança de operações

18

## Restrições múltiplas

```
vSort :: (Ord t, Visible t) => [t] -> String
vSort = toString.iSort
```

```
instance (Eq t, Eq u) => Eq (t,u) where
  (a,b) == (c,d) = (a == c && b == d)
```

```
class (Ord t, Visible t) => OrdVis t
```

Herança múltipla

19

## Classes predefinidas

```
show :: (Show t) => t -> String  
read :: (Read t) => String -> t
```

## Bibliografia

Slides elaborados a partir de originais por André Santos e Fernando Castor

[1] Simon Thompson.

***Haskell: the craft of functional programming.***

Addison-Wesley, terceira edition, Julho 2011.