Lenguajes, Tecnologías y Paradigmas de la programación (LTP)

Práctica 6: Módulos y Polimorfismo en Haskell

(Parte 1: Módulos y Polimorfismo paramétrico)

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Sergio Pérez
serperu@dsic.upv.es

SESIÓN 1: Módulos y Polimorfismo Paramétrico

OBJETIVOS DE LA PRÁCTICA

Trabajar con operaciones básicas sobre **módulos**:

- Importación de módulos
- Listas de exportación
- Importaciones *qualified*

Polimorfismo paramétrico:

- Aprender para qué sirven las clases de tipos

Módulos (Exportación/Importación)

- Listas de exportación:

```
module Sphere where
    …
```

Módulos (Exportación/Importación)

- Listas de exportación:

```
module Sphere where
    …                          ➙ Todas las funciones
```

Módulos (Exportación/Importación)

- Listas de exportación:

```
module Sphere where
    …                          → Todas las funciones
module Sphere (area, volume) where
    …
```

Módulos (Exportación/Importación)

- Listas de exportación:

```
module Sphere where
    …                    → Todas las funciones
module Sphere (area, volume) where
    …                    → Funciones area y volume
```

Módulos (Exportación/Importación)

- Listas de exportación:

```
module Sphere where
    …                    → Todas las funciones
module Sphere (area, volume) where
    …                    → Funciones area y volume
```

- Importación de módulos:

```
import Sphere
```

Módulos (Exportación/Importación)

- Importaciones qualified:

```
module Circle (area) where
 area :: Float -> Float
 area r = pi * ((^2) r)
```

Módulos (Exportación/Importación)

- Importaciones qualified:

```
module Circle (area) where
 area :: Float -> Float
 area r = pi * ((^2) r)

module Square (area) where
 area :: Float -> Float
 area s = ((^2) s)
```

Módulos (Exportación/Importación)

- Importaciones qualified:

```
module Circle (area) where
 area :: Float -> Float
 area r = pi * ((^2) r)

module Square (area) where
 area :: Float -> Float
 area s = ((^2) s)
```

```
module Test where
 import qualified Circle
 import qualified Square
```

Módulos (Exportación/Importación)

- Importaciones qualified:

```
module Circle (area) where
 area :: Float -> Float
 area r = pi * ((^2) r)

module Square (area) where
 area :: Float -> Float
 area s = ((^2) s)
```

```
module Test where
 import qualified Circle
 import qualified Square

 Circle.area 4.5
```

Módulos (Exportación/Importación)

- Importaciones qualified:

```
module Circle (area) where
 area :: Float -> Float
 area r = pi * ((^2) r)

module Square (area) where
 area :: Float -> Float
 area s = ((^2) s)
```

```
module Test where
 import qualified Circle
 import qualified Square

 Circle.area 4.5
 Square.area 2.5
```

Queue Module (Tipo recursivo)

```haskell
module Queue (Queue, empty, enqueue, dequeue, first, isEmpty, size) where
  data Queue a = EmptyQueue | Item a (Queue a)

  empty = EmptyQueue

  enqueue x EmptyQueue = Item x EmptyQueue
  enqueue x (Item a q) = Item a (enqueue x q)

  dequeue (Item _ q) = q

  first (Item a _) = a

  isEmpty EmptyQueue = True
  isEmpty _ = False

  size EmptyQueue = 0
  size (Item _ q) = 1 + size q
```

Queue Module (Tipo recursivo)

```haskell
module Queue (Queue, empty, enqueue, dequeue, first, isEmpty, size) where
  data Queue a = EmptyQueue | Item a (Queue a)

  empty = EmptyQueue          → Crea una cola vacía (EmptyQueue)

  enqueue x EmptyQueue = Item x EmptyQueue
  enqueue x (Item a q) = Item a (enqueue x q)

  dequeue (Item _ q) = q

  first (Item a _) = a

  isEmpty EmptyQueue = True
  isEmpty _ = False

  size EmptyQueue = 0
  size (Item _ q) = 1 + size q
```

Queue Module (Tipo recursivo)

```haskell
module Queue (Queue, empty, enqueue, dequeue, first, isEmpty, size) where
  data Queue a = EmptyQueue | Item a (Queue a)

  empty = EmptyQueue          -- Crea una cola vacía (EmptyQueue)

  enqueue x EmptyQueue = Item x EmptyQueue
  enqueue x (Item a q) = Item a (enqueue x q)  -- Encola x en la parte interna

  dequeue (Item _ q) = q

  first (Item a _) = a

  isEmpty EmptyQueue = True
  isEmpty _ = False

  size EmptyQueue = 0
  size (Item _ q) = 1 + size q
```

Queue Module (Tipo recursivo)

```
enqueue x EmptyQueue = Item x EmptyQueue
enqueue x (Item a q) = Item a (enqueue x q)
```

Queue Module (Tipo recursivo)

```
enqueue x EmptyQueue = Item x EmptyQueue
enqueue x (Item a q) = Item a (enqueue x q)
```

```
Prelude> enqueue 5 (Item 3 EmptyQueue)
```

Queue Module (Tipo recursivo)

```
enqueue x EmptyQueue = Item x EmptyQueue
enqueue x (Item a q) = Item a (enqueue x q)
```

```
Prelude> enqueue 5 (Item 3 EmptyQueue)
```

```
x = 5; a = 3; q = EmptyQueue
```

Queue Module (Tipo recursivo)

```
enqueue x EmptyQueue = Item x EmptyQueue
enqueue x (Item a q) = Item a (enqueue x q)
```

```
Prelude> enqueue 5 (Item 3 EmptyQueue)
```

```
x = 5; a = 3; q = EmptyQueue
```

¿Qué cola devuelvo?

Queue Module (Tipo recursivo)

```
enqueue x EmptyQueue = Item x EmptyQueue
enqueue x (Item a q) = Item a (enqueue x q)
```

```
Prelude> enqueue 5 (Item 3 EmptyQueue)
```

```
x = 5; a = 3; q = EmptyQueue
```

¿Qué cola devuelvo?

```
Item 3 (enqueue 5 EmptyQueue)
```

Queue Module (Tipo recursivo)

```
enqueue x EmptyQueue = Item x EmptyQueue
enqueue x (Item a q) = Item a (enqueue x q)
```

```
Prelude> enqueue 5 (Item 3 EmptyQueue)
```

```
x = 5; a = 3; q = EmptyQueue
```

¿Qué cola devuelvo?

```
Item 3 (enqueue 5 EmptyQueue) →
```

```
Item 3
```

Queue Module (Tipo recursivo)

```
enqueue x EmptyQueue = Item x EmptyQueue
enqueue x (Item a q) = Item a (enqueue x q)
```

```
Prelude> enqueue 5 (Item 3 EmptyQueue)
```

```
x = 5; a = 3; q = EmptyQueue
```

¿Qué cola devuelvo?

```
Item 3 (enqueue 5 EmptyQueue) →
Item 3 (Item 5 EmptyQueue)
```

Queue Module (Tipo recursivo)

```haskell
module Queue (Queue, empty, enqueue, dequeue, first, isEmpty, size) where
  data Queue a = EmptyQueue | Item a (Queue a)

  empty = EmptyQueue          -- Crea una cola vacía (EmptyQueue)

  enqueue x EmptyQueue = Item x EmptyQueue
  enqueue x (Item a q) = Item a (enqueue x q)  -- Encola x en la parte interna

  dequeue (Item _ q) = q

  first (Item a _) = a

  isEmpty EmptyQueue = True
  isEmpty _ = False

  size EmptyQueue = 0
  size (Item _ q) = 1 + size q
```

Queue Module (Tipo recursivo)

```haskell
module Queue (Queue, empty, enqueue, dequeue, first, isEmpty, size) where
  data Queue a = EmptyQueue | Item a (Queue a)

  empty = EmptyQueue          -- Crea una cola vacía (EmptyQueue)

  enqueue x EmptyQueue = Item x EmptyQueue
  enqueue x (Item a q) = Item a (enqueue x q)   -- Encola x en la parte interna

  dequeue (Item _ q) = q      -- Elimina el elemento más externo (el primero que se insertó)

  first (Item a _) = a

  isEmpty EmptyQueue = True
  isEmpty _ = False

  size EmptyQueue = 0
  size (Item _ q) = 1 + size q
```

Queue Module (Tipo recursivo)

```haskell
module Queue (Queue, empty, enqueue, dequeue, first, isEmpty, size) where
  data Queue a = EmptyQueue | Item a (Queue a)

  empty = EmptyQueue              -- Crea una cola vacía (EmptyQueue)

  enqueue x EmptyQueue = Item x EmptyQueue
  enqueue x (Item a q) = Item a (enqueue x q)   -- Encola x en la parte interna

  dequeue (Item _ q) = q          -- Elimina el elemento más externo (el primero que se insertó)

  first (Item a _) = a            -- Devuelve el elemento más externo (el primero que se insertó)

  isEmpty EmptyQueue = True
  isEmpty _ = False

  size EmptyQueue = 0
  size (Item _ q) = 1 + size q
```

Queue Module (Tipo recursivo)

```haskell
module Queue (Queue, empty, enqueue, dequeue, first, isEmpty, size) where
  data Queue a = EmptyQueue | Item a (Queue a)

  empty = EmptyQueue              -- Crea una cola vacía (EmptyQueue)

  enqueue x EmptyQueue = Item x EmptyQueue
  enqueue x (Item a q) = Item a (enqueue x q)   -- Encola x en la parte interna

  dequeue (Item _ q) = q          -- Elimina el elemento más externo (el primero que se insertó)

  first (Item a _) = a            -- Devuelve el elemento más externo (el primero que se insertó)

  isEmpty EmptyQueue = True
  isEmpty _ = False               -- Devuelve True si la cola esta vacía

  size EmptyQueue = 0
  size (Item _ q) = 1 + size q
```

Queue Module (Tipo recursivo)

```
module Queue (Queue, empty, enqueue, dequeue, first, isEmpty, size) where
  data Queue a = EmptyQueue | Item a (Queue a)

  empty = EmptyQueue              → Crea una cola vacía (EmptyQueue)

  enqueue x EmptyQueue = Item x EmptyQueue
  enqueue x (Item a q) = Item a (enqueue x q)  → Encola x en la parte interna

  dequeue (Item _ q) = q          → Elimina el elemento más externo (el primero que se insertó)

  first (Item a _) = a            → Devuelve el elemento más externo (el primero que se insertó)

  isEmpty EmptyQueue = True
  isEmpty _ = False               → Devuelve True si la cola esta vacía

  size EmptyQueue = 0
  size (Item _ q) = 1 + size q    → Devuelve el tamaño de la cola
```

Queue Module (2 listas)

```haskell
module Queue where
  data Queue a = Queue [a] [a]

  empty = Queue [] []

  enqueue y (Queue xs ys) = Queue xs (y:ys)

  dequeue (Queue (x:xs) ys) = Queue xs ys
  dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])

  first (Queue (x:xs) ys) = x
  first (Queue [] ys) = head (reverse ys)

  isEmpty (Queue [] []) = True
  isEmpty _ = False

  size (Queue a b) = length a + length b
```

Queue Module (2 listas)

```haskell
module Queue where
  data Queue a = Queue [a] [a]

  empty = Queue [] []          -- Crea una cola vacía (Queue [] [])

  enqueue y (Queue xs ys) = Queue xs (y:ys)

  dequeue (Queue (x:xs) ys) = Queue xs ys
  dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])

  first (Queue (x:xs) ys) = x
  first (Queue [] ys) = head (reverse ys)

  isEmpty (Queue [] []) = True
  isEmpty _ = False

  size (Queue a b) = length a + length b
```

Queue Module (2 listas)

```haskell
module Queue where
  data Queue a = Queue [a] [a]

  empty = Queue [] []          -- Crea una cola vacía (Queue [] [])

  enqueue y (Queue xs ys) = Queue xs (y:ys)   -- Mete el elemento en la cabeza de ys

  dequeue (Queue (x:xs) ys) = Queue xs ys
  dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])

  first (Queue (x:xs) ys) = x
  first (Queue [] ys) = head (reverse ys)

  isEmpty (Queue [] []) = True
  isEmpty _ = False

  size (Queue a b) = length a + length b
```

Queue Module (2 listas)

```
module Queue where
  data Queue a = Queue [a] [a]

  empty = Queue [] []          → Crea una cola vacía (Queue [] [])

  enqueue y (Queue xs ys) = Queue xs (y:ys)  → Mete el elemento en la cabeza de ys

  dequeue (Queue (x:xs) ys) = Queue xs ys  → Saca el primer elemento de la primera lista
  dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])

  first (Queue (x:xs) ys) = x
  first (Queue [] ys) = head (reverse ys)

  isEmpty (Queue [] []) = True
  isEmpty _ = False

  size (Queue a b) = length a + length b
```

Queue Module (2 listas)

```
module Queue where
  data Queue a = Queue [a] [a]

  empty = Queue [] []          → Crea una cola vacía (Queue [] [])

  enqueue y (Queue xs ys) = Queue xs (y:ys)  → Mete el elemento en la cabeza de ys

  dequeue (Queue (x:xs) ys) = Queue xs ys  → Saca el primer elemento de la primera lista
  dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])  → Si la 1ª esta vacía, la rellena

  first (Queue (x:xs) ys) = x
  first (Queue [] ys) = head (reverse ys)

  isEmpty (Queue [] []) = True
  isEmpty _ = False

  size (Queue a b) = length a + length b
```

Queue Module (2 listas)

```
enqueue y (Queue xs ys) = Queue xs (y:ys)

dequeue (Queue (x:xs) ys) = Queue xs ys
dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])
```

Queue Module (2 listas)

```
enqueue y (Queue xs ys) = Queue xs (y:ys)

dequeue (Queue (x:xs) ys) = Queue xs ys
dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])
```

1) `enqueue 6 (Queue [] [])`

Queue Module (2 listas)

```
enqueue y (Queue xs ys) = Queue xs (y:ys)

dequeue (Queue (x:xs) ys) = Queue xs ys
dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])
```

1) `enqueue 6 (Queue [] [])` → `Queue [] [6]`

Queue Module (2 listas)

```
enqueue y (Queue xs ys) = Queue xs (y:ys)

dequeue (Queue (x:xs) ys) = Queue xs ys
dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])
```

1) `enqueue 6 (Queue [] [])` → `Queue [] [6]`

2) `enqueue 9 (Queue [] [6])`

Queue Module (2 listas)

```
enqueue y (Queue xs ys) = Queue xs (y:ys)

dequeue (Queue (x:xs) ys) = Queue xs ys
dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])
```

1) `enqueue 6 (Queue [] []) → Queue [] [6]`

2) `enqueue 9 (Queue [] [6]) → Queue [] [9,6]`

Queue Module (2 listas)

```
enqueue y (Queue xs ys) = Queue xs (y:ys)

dequeue (Queue (x:xs) ys) = Queue xs ys
dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])
```

1) `enqueue 6 (Queue [] []) → Queue [] [6]`

2) `enqueue 9 (Queue [] [6]) → Queue [] [9,6]`

3) `dequeue (Queue [] [9,6])`

Queue Module (2 listas)

```
enqueue y (Queue xs ys) = Queue xs (y:ys)

dequeue (Queue (x:xs) ys) = Queue xs ys
dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])
```

1) `enqueue 6 (Queue [] [])` → `Queue [] [6]`

2) `enqueue 9 (Queue [] [6])` → `Queue [] [9,6]`

3) `dequeue (Queue [] [9,6])` → **¿Qué debe eliminar?**

Queue Module (2 listas)

```
enqueue y (Queue xs ys) = Queue xs (y:ys)

dequeue (Queue (x:xs) ys) = Queue xs ys
dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])
```

1) `enqueue 6 (Queue [] [])` → `Queue [] [6]`

2) `enqueue 9 (Queue [] [6])` → `Queue [] [9,6]`

3) `dequeue (Queue [] [9,6])` → **¿Qué debe eliminar?** **6**

Queue Module (2 listas)

```
enqueue y (Queue xs ys) = Queue xs (y:ys)

dequeue (Queue (x:xs) ys) = Queue xs ys
dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])
```

1) `enqueue 6 (Queue [] [])` → `Queue [] [6]`

2) `enqueue 9 (Queue [] [6])` → `Queue [] [9,6]`

3) `dequeue (Queue [] [9,6])` → **¿Qué debe eliminar?** **6**

→ `dequeue (Queue (reverse [9,6]) [])`

Queue Module (2 listas)

```
enqueue y (Queue xs ys) = Queue xs (y:ys)

dequeue (Queue (x:xs) ys) = Queue xs ys
dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])
```

1) `enqueue 6 (Queue [] [])` → `Queue [] [6]`

2) `enqueue 9 (Queue [] [6])` → `Queue [] [9,6]`

3) `dequeue (Queue [] [9,6])` → **¿Qué debe eliminar?** **6**

→ `dequeue (Queue (reverse [9,6]) [])`

→ `dequeue (Queue [6,9] [])`

Queue Module (2 listas)

```
enqueue y (Queue xs ys) = Queue xs (y:ys)

dequeue (Queue (x:xs) ys) = Queue xs ys
dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])
```

1) `enqueue 6 (Queue [] [])` → `Queue [] [6]`

2) `enqueue 9 (Queue [] [6])` → `Queue [] [9,6]`

3) `dequeue (Queue [] [9,6])` → **¿Qué debe eliminar?** **6**

→ `dequeue (Queue (reverse [9,6]) [])`

→ `dequeue (Queue [6,9] [])` → **6**

Queue Module (2 listas)

```
module Queue where
  data Queue a = Queue [a] [a]

  empty = Queue [] []          → Crea una cola vacía (Queue [] [])

  enqueue y (Queue xs ys) = Queue xs (y:ys)  → Mete el elemento en la cabeza de ys

  dequeue (Queue (x:xs) ys) = Queue xs ys  → Saca el primer elemento de la primera lista
  dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])  → Si la 1ª esta vacía, la rellena

  first (Queue (x:xs) ys) = x  → Devuelve el primer elemento de la primera lista
  first (Queue [] ys) = head (reverse ys)  → Si la 1ª lista es vacía, saca el último de la 2ª

  isEmpty (Queue [] []) = True
  isEmpty _ = False

  size (Queue a b) = length a + length b
```

Queue Module (2 listas)

```
module Queue where
  data Queue a = Queue [a] [a]

  empty = Queue [] []                  → Crea una cola vacía (Queue [] [])

  enqueue y (Queue xs ys) = Queue xs (y:ys)  → Mete el elemento en la cabeza de ys

  dequeue (Queue (x:xs) ys) = Queue xs ys  → Saca el primer elemento de la primera lista
  dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])  → Si la 1ª esta vacía, la rellena

  first (Queue (x:xs) ys) = x  → Devuelve el primer elemento de la primera lista
  first (Queue [] ys) = head (reverse ys)  → Si la 1ª lista es vacía, saca el último de la 2ª

  isEmpty (Queue [] []) = True
  isEmpty _ = False                    → Devulve si la cola es vacía (ambas listas vacias)

  size (Queue a b) = length a + length b
```

Queue Module (2 listas)

```
module Queue where
  data Queue a = Queue [a] [a]

  empty = Queue [] []                    → Crea una cola vacía (Queue [] [])

  enqueue y (Queue xs ys) = Queue xs (y:ys)  → Mete el elemento en la cabeza de ys

  dequeue (Queue (x:xs) ys) = Queue xs ys  → Saca el primer elemento de la primera lista
  dequeue (Queue [] ys) = dequeue (Queue (reverse ys) [])  → Si la 1ª esta vacía, la rellena

  first (Queue (x:xs) ys) = x  → Devuelve el primer elemento de la primera lista
  first (Queue [] ys) = head (reverse ys)  → Si la 1ª lista es vacía, saca el último de la 2ª

  isEmpty (Queue [] []) = True
  isEmpty _ = False                      → Devulve si la cola es vacía (ambas listas vacias)

  size (Queue a b) = length a + length b  → Devuelve el total de elementos de ambas listas
```

Polimorfismo paramétrico (clases de tipos)

- Longitud de una lista:

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

Polimorfismo paramétrico (clases de tipos)

- Longitud de una lista:

```
length :: [a] -> Int
length [] = 0                    → No utiliza los elementos ni su tipo a
length (x:xs) = 1 + length xs
```

Polimorfismo paramétrico (clases de tipos)

- Longitud de una lista:

```
length :: [a] -> Int
length [] = 0                          → No utiliza los elementos ni su tipo a
length (x:xs) = 1 + length xs
```

- Operación “(==)”

Polimorfismo paramétrico (clases de tipos)

- Longitud de una lista:

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

→ No utiliza los elementos ni su tipo a

- Operación “(==)”

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

Polimorfismo paramétrico (clases de tipos)

- Longitud de una lista:

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

→ No utiliza los elementos ni su tipo a

- Operación “(==)”

```
(==) :: [a] -> [a] -> Bool
[]     == []     = True
```

Polimorfismo paramétrico (clases de tipos)

- Longitud de una lista:

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

→ No utiliza los elementos ni su tipo a

- Operación “(==)”

```
(==) :: [a] -> [a] -> Bool
[]     == []     = True
[]     == (x:xs) = False
```

Polimorfismo paramétrico (clases de tipos)

- Longitud de una lista:

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

→ No utiliza los elementos ni su tipo a

- Operación “(==)”

```
(==) :: [a] -> [a] -> Bool
[]     == []     = True
[]     == (x:xs) = False
(x:xs) == []     = False
```

Polimorfismo paramétrico (clases de tipos)

- Longitud de una lista:

```
length :: [a] -> Int
length [] = 0                          → No utiliza los elementos ni su tipo a
length (x:xs) = 1 + length xs
```

- Operación “(==)”

```
(==) :: [a] -> [a] -> Bool
[]     == []     = True
[]     == (x:xs) = False
(x:xs) == []     = False
(x:xs) == (y:ys) = x==y && xs==ys
```

Polimorfismo paramétrico (clases de tipos)

- Longitud de una lista:

```
length :: [a] -> Int
length [] = 0                    → No utiliza los elementos ni su tipo a
length (x:xs) = 1 + length xs
```

- Operación “(==)”

```
(==) :: [a] -> [a] -> Bool
[]     == []     = True
[]     == (x:xs) = False          → Necesita comparar x e y, de tipo a
(x:xs) == []     = False
(x:xs) == (y:ys) = x==y && xs==ys
```

Polimorfismo paramétrico (clases de tipos)

- Para poder indicar la restricción de que el tipo “a” debe admitir la comparación, Haskell utiliza las denominadas ***clases de tipos***:

Polimorfismo paramétrico (clases de tipos)

- Para poder indicar la restricción de que el tipo “a” debe admitir la comparación, Haskell utiliza las denominadas ***clases de tipos***:

  La clase de tipos ***Eq*** representa los tipos que tienen definidas las operaciones “==” y “/=”.

Polimorfismo paramétrico (clases de tipos)

- Para poder indicar la restricción de que el tipo “a” debe admitir la comparación, Haskell utiliza las denominadas ***clases de tipos***:

La clase de tipos ***Eq*** representa los tipos que tienen definidas las operaciones “==” y “/=”.

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

…

Polimorfismo paramétrico (clases de tipos)

- Para poder indicar la restricción de que el tipo “a” debe admitir la comparación, Haskell utiliza las denominadas ***clases de tipos***:

La clase de tipos ***Eq*** representa los tipos que tienen definidas las operaciones “==” y “/=”.

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

…

Polimorfismo paramétrico (clases de tipos)

- Para poder indicar la restricción de que el tipo “a” debe admitir la comparación, Haskell utiliza las denominadas ***clases de tipos***:

La clase de tipos ***Eq*** representa los tipos que tienen definidas las operaciones “==” y “/=”.

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

…

Similar a las “interfaces” en **Java** (expresa un comportamiento para el tipo de dato a)

Ejercicios

- Ejercicios Parte 1:
  - Ejercicios 1 - 5

Polimorfismo paramétrico (clases de tipos)

```
data Queue a = EmptyQueue | Item a (Queue a)
```

Imaginad la cola (1,5,9)

Polimorfismo paramétrico (clases de tipos)

```
data Queue a = EmptyQueue | Item a (Queue a)
```

Imaginad la cola (1,5,9)

Si añado el deriving Show Haskell para enseñarme los datos de tipo *Queue* por pantalla:

Polimorfismo paramétrico (clases de tipos)

```
data Queue a = EmptyQueue | Item a (Queue a)
```

Imaginad la cola (1,5,9)

Si añado el deriving Show Haskell para enseñarme los datos de tipo *Queue* por pantalla:

```
data Queue a = EmptyQueue | Item a (Queue a) deriving Show
```

```
Item 1 (Item 5 (Item 9 EmptyQueue))
```

Polimorfismo paramétrico (clases de tipos)

¿Y si quiero una representación más legible?

Ejemplo: `"<- 1 <- 5 <- 9 <-"`

Polimorfismo paramétrico (clases de tipos)

¿Y si quiero una representación más legible?

Ejemplo: `"<- 1 <- 5 <- 9 <-"`

¡¡POS LA IMPLEMENTAS TÚ!!

Polimorfismo paramétrico (clases de tipos)

¿Y si quiero una representación más legible?

Ejemplo: `"<- 1 <- 5 <- 9 <-"`

Diré que esta clase es una instancia de la clase de tipos **Show**

(Al final del fichero donde haya definido este tipo de datos)

Polimorfismo paramétrico (clases de tipos)

¿Y si quiero una representación más legible?

Ejemplo: `"<- 1 <- 5 <- 9 <-"`

Diré que esta clase es una instancia de la clase de tipos **Show**

(Al final del fichero donde haya definido este tipo de datos)

```
instance                    Show (Queue a) where
```

Polimorfismo paramétrico (clases de tipos)

¿Y si quiero una representación más legible?

Ejemplo: `"<- 1 <- 5 <- 9 <-"`

Diré que esta clase es una instancia de la clase de tipos **Show**

(Al final del fichero donde haya definido este tipo de datos)

```
instance                Show (Queue a) where
  show EmptyQueue = " <- "
```

Polimorfismo paramétrico (clases de tipos)

¿Y si quiero una representación más legible?

Ejemplo: `"<- 1 <- 5 <- 9 <-"`

Diré que esta clase es una instancia de la clase de tipos **Show**

(Al final del fichero donde haya definido este tipo de datos)

```
instance                Show (Queue a) where
  show EmptyQueue = " <- "
  show (Item x y) =
```

Polimorfismo paramétrico (clases de tipos)

¿Y si quiero una representación más legible?

Ejemplo: `"<- 1 <- 5 <- 9 <-"`

Diré que esta clase es una instancia de la clase de tipos **Show**

(Al final del fichero donde haya definido este tipo de datos)

```
instance                Show (Queue a) where
  show EmptyQueue = " <- "
  show (Item x y) = " <- " ++
```

Polimorfismo paramétrico (clases de tipos)

¿Y si quiero una representación más legible?

Ejemplo: `"<- 1 <- 5 <- 9 <-"`

Diré que esta clase es una instancia de la clase de tipos **Show**

(Al final del fichero donde haya definido este tipo de datos)

```
instance                Show (Queue a) where
  show EmptyQueue = " <- "
  show (Item x y) = " <- " ++ (show x)
```

Polimorfismo paramétrico (clases de tipos)

¿Y si quiero una representación más legible?

Ejemplo: `"<- 1 <- 5 <- 9 <-"`

Diré que esta clase es una instancia de la clase de tipos **Show**

(Al final del fichero donde haya definido este tipo de datos)

```
instance                Show (Queue a) where
  show EmptyQueue = " <- "
  show (Item x y) = " <- " ++ (show x) ++ (show y)
```

Polimorfismo paramétrico (clases de tipos)

¿Y si quiero una representación más legible?

Ejemplo: `"<- 1 <- 5 <- 9 <-"`

Diré que esta clase es una instancia de la clase de tipos **Show**

```
data Queue a = EmptyQueue | Item a (Queue a)
```

```
instance                  Show (Queue a) where
  show EmptyQueue = " <- "
  show (Item x y) = " <- " ++ (show x) ++ (show y)
```

Polimorfismo paramétrico (clases de tipos)

¿Y si quiero una representación más legible?

Ejemplo: `"<- 1 <- 5 <- 9 <-"`

Diré que esta clase es una instancia de la clase de tipos **Show**

```
data Queue a = EmptyQueue | Item a (Queue a)
```

```
instance (Show a) => Show (Queue a) where
  show EmptyQueue = " <- "
  show (Item x y) = " <- " ++ (show x) ++ (show y)
```