typeclasses.lhs

```
> {-# LANGUAGE FlexibleInstances #-}
> {-# LANGUAGE UndecidableInstances #-}
```

^ Здесь указаны несколько расширений языка, которые я поясню позже. Расширения включают фичи языка, которые не вошли в стандарт Haskell'2010, но уже реализованы в компиляторе.

```
> module Typeclasses where
```

Классы типов в Haskell

Раньше мы рассматривали только мономорфные функции вида

```
f :: Int -> Int
f = (+ 10)
```

и неограниченный параметрический полиморфизм, когда мы ничего не знаем о типе

```
g :: a -> (a, a)
g x = (x, x)
```

Но иногда хочется сохранить полиморфизм, но при этом знать про типы хоть немного. Тут нам на помощь приходят классы типов.

Класс типов описывает некие общие свойства, которыми обладают некоторые типы. Имея некий класс, мы можем наложить ограничение на конкретную переменную типа в конкретном месте кода. После наложения ограничения подставить в переменную будет можно только те типы, которые реализуют нужное поведение (говорят *инстанциируют класс*).

Пример класса:

```
> -- Класс "Имеет имя"
> class Equable a where
>   --         ^ тип, который входит в класс

>   -- Функции, объявляемые в классе, называются "методами класса".
>   -- Методы могут иметь умолчательные реализации и быть реализованы
>   -- друг через друга.

>   (===) :: a -> a -> Bool
>   -- здесь a ^ -- это та же переменная, что и в заголовке класса,
>   -- т.е. тот же самый тип
```

А это пример кода, его использующего:

```
> data Thing = This | That

> instance Equable Thing where
>   --           ^ вместо "a" подставлен тип, для которого пишется инстанс.
>   This === This = True
>   That === That = True
>   _    === _    = False

>   -- метод "!==" я реализовывать не стал, поэтому для этого типа будет
>   -- использоваться умолчательная реализация.

> allTheSame
>   :: Equable a =>
>   -- эта часть называется ограничением (constraint). Здесь говорится,
>   -- что "a" должен иметь инстанс класса Equable.
>      [a] -> Bool
> allTheSame [] = False
> allTheSame (x:xs) = foldl (\acc y -> acc && x === y) True xs
```

Инстансы можно писать постфактум. Вы можете инстанциировать чужие классы для своих типов или же свои классы для чужих типов. Но компилятор будет следить за **когерентностью** инстансов: для конкретного класса и конкретного типа может быть использован только один инстанс – один на всю вашу программу. При импортировании типа или класса всегда импортируются все инстансы, хотим мы того или нет. Это тоже сделано для того, чтобы нельзя было в разных модулях использовать разные инстансы для одной и той же пары *класс-тип*.

При описании класса можно наложить ограничение, требующее, чтобы тип, претендующий на вхождение в класс, уже инстанциировал какие-то другие классы. Обычно это делается для того, чтобы в дефолтных реализациях методов описываемого класса можно было использовать методы других классов.

Вот пример такого класса:

```
> class Equable a => Comparable a where

>   (?<), (?>), (?<=), (?>=) :: a -> a -> Bool

>   -- В этом контексте я знаю, что x будет Equable, поэтому могу
>   -- вызывать для него (===).
>   x ?>  y = not $ x ?<= y
>   x ?<  y = not $ x ?>= y
>   x ?<= y = x === y || x ?< y
>   x ?>= y = x === y || x ?> y
>   -- здесь опять всё выражено через всё, поэтому если не заставить
>   -- переопределить хоть что-то, то код зациклится при первом вызове.
>   -- Для затребования реализации разумного минимума существует прагма:
>   {-# MINIMAL (?>) | (?<) #-}
```

В инстансах тоже можно указывать ограничения, если мы хотим предоставить реализацию не для конкретного типа, а для полиморфного. Более того, лишь указание ограничений и позволит нам такой инстанс сделать! Представим, что мы сделали такой инстанс:

```
instance Equable a where
  _ === _ = False
```

Этот инстанс реализует класс для абсолютно любого типа. А значит никаких других инстансов сделать будет нельзя, ибо когерентность! И даже так сделать нельзя:

```
instance Equable a => Comparable a where
```

У этого инстанса проблема в том, что информации недостаточно, чтобы остановить поиск инстанса для конкретного типа, а значит поиск инстанса зациклится! Так произойдёт потому, что под этот инстанс подойдут все типы, потенциально подходящие под ограничения самого класса, а значит этот инстанс не добавляет никакой конкретики.

GHC старается найти подходящий инстанс максимально быстро, поэтому запрещает писать такие инстансы, которые поломают поиск. Чтобы сказать ‘я сам умный!’ и местно снять сие ограничение, включают расширение UndecidableInstances

Рассмотрим пример правильно ограниченного инстанса. В примере я использую класс Eq из стандартной поставки Haskell, реализация которого выглядит примерно так (я опустил часть методов):

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

Именно этот класс отвечает за проверку на равенство в реальном коде!

Итак, пример:

```
> instance Eq a => Equable a where
>   -- "инстанс для тех типов, которые ещё и Show реализуют"

>   (===) = (==)
>   -- тут я редуцировал "a === b = a == b" до уравнивания методов :)

> numberIsPositive :: Int -> Bool
> numberIsPositive x = abs x === x
> -- заметьте, я не использовать проверку на "?> 0", потому что
> -- для Int нет инстанса Comparable. А вот "===" работает потому,
> -- что Int инстанциирует Eq!

> -- Важно: это очень синтетический пример, проверять числа на позитивность
> -- таким образом не следует :)
```

Описанные здесь классы Equable и Comparable являются грубыми клонами встроенных классов Eq и Ord, но зато хорошо демонстрируют наследование поведения. В реальном коде вы будете использовать именно Eq и Ord.

Задание

Для практики реализуйте инстансы

```
instance Eq Thing where
instance Ord Thing where
```

потом добавьте тип

```
> data Pair a = Pair a a
```

и реализуйте инстансы

```
instance Eq Pair where
instance Ord Pair where
```

В экспериментах в REPL вам потребуются также инстансы класса Show для новых типов. Реализуйте вручную инстансы

```
instance Show Thing where
instance Show a => Show (Pair a) where
```

Документация по классам:

- Eq
- Ord
- Show