



Functional Programming in Haskell

[The University of Glasgow](#)

[Get Unlimited learning](#)

[€23.99/month](#)

[TP](#)

- [Your learning](#)
- [Wishlist](#)
- [Profile](#)
- [Settings](#)
- [Account](#)
- [Sign out](#)

[Get Unlimited learning](#)

[€23.99/month](#)

- [To do](#)
- [Conversations](#)
- [Progress](#)

Week 1

Haskell First Steps

Introduction

1.1

[Welcome to the Course](#)

[Video](#)

Haskell Basics: Expressions and Equations

1.2

[Basic Elements By Example](#)[Video](#)

1.3

[Introduction to Expressions and Equations](#)[Article](#)

1.4

[Do it Yourself: Expressions, Functions and Equations](#)[Exercise](#)

1.5

[Test Your Understanding](#)[Quiz](#)

1.6

[Summary](#)[Article](#)

Haskell Basics: Reduction, Functions and Lists

1.7

[More Basic Elements by Example](#)[Video](#)

6.3

You've completed 2 steps in Week 6

[Functions and Lists](#)[Do it Yourself: Functions and Lists](#)

© Wim Vanderbauwhede

[Exercise](#)**Type Classes in more Detail**

[2.10 Comments](#)[Test Your Understanding](#)

Type class and instance declarations

[Quiz](#)

Defining type classes

1.11

- A *type class* is a set of types for which some operations are defined.
- Haskell has some standard type classes that are defined in the Standard Prelude.
- You can also define your own.

[Summary](#)[Article](#)

A type for bright colors

Suppose we're computing with colors. Here's a type, and a couple of functions.

1.12

```
data Bright
```

[Recommended Reading](#)

```
  = Red
```

[Video](#)

```
  deriving (Read, Show)
```

```
darkBright :: Bright -> Bool
darkBright Blue = True
darkBright Red  = False
```

[Spot the Difference](#)

```
lightenBright :: Bright -> Bright
```

[Discussion](#)

```
lightenBright Blue = Red
lightenBright Red  = Red
```

1.14

A type for milder colors

[End of Week 1](#)

Now, suppose we have a different type that needs similar functions.

[Video](#)

```
data Pastel
```

```
  = Turquoise
```

```
  = Tan
```

```
  deriving (Read, Show)
```

Haskell Building blocks

```
darkPastel :: Pastel -> Bool
```

```
darkPastel Turquoise = True
```

```
darkPastel Tan       = False
```

```
lightenPastel :: Pastel -> Pastel
```

```
lightenPastel Turquoise = Tan
```

2.1

```
lightenPastel Tan       = Tan
```

[Welcome to week 2](#)

Defining a type class

[Video](#)

- Both of our color types have functions to decide whether it's dark, or to lighten it.
- We can define a class *Color* and its corresponding functions.

[Do it Yourself: Boolean Values and Expressions](#)[Exercise](#)

```
class Color a where
```

```
  dark :: a -> Bool
```

2.3

```
  lighten :: a -> a
```

This says

[Zip that List](#)

[Video](#) `Color` is a type class

- The type variable a stands for a particular type that is in the class `Color`
- 2.4 • For any type a in `Color`, there are two functions you can use: `dark` and `lighten`, with the specified types.

[Do it Yourself: Logical Thinking](#)

Defining instances for the type class

[Exercise](#)

- 2.5 • An *instance* declaration says that a type is a member of a type class.
- When you declare an instance, you need to define the class functions.
- [Nothing Trusty](#) says that the type `Bright` is in the class `Color`, and for that instance, the `dark` function is actually `darkBright`.

[Quiz](#)

```
instance Color Bright where
  Input dark = darkBright
  lighten = lightenBright
```

- 2.6 • Similarly, we can declare that `Pastel` is in `Color`, but it has different functions to implement the class operations.

[Why I/O?](#)

[Video](#) `instance Color Pastel where`
`dark = darkPastel`
`lighten = lightenPastel`

2.7

Predefined type classes

[Do it Yourself: Input/Output](#)

[Haskell](#) provides several standard type classes. We have a look at two of them: `Num` and `Show`.

2.8

The Num class

[I/O and a First Encounter with Monads](#)

- `Num` is the class of numeric types.
- Here is (part of) its class declaration:

[Article](#)

```
class Num a where
  (+), (-), (*) :: a -> a -> a
```

Num instances

[Installing Haskell for Yourself](#)

- There are many numeric types; two of them are `Int` and `Double`.
- There are primitive monomorphic functions that perform arithmetic on these types (these aren't the real names):

[Article](#)

```
addInt, subInt, mulInt :: Int -> Int -> Int
addDbl, subDbl, mulDbl :: Double -> Double -> Double
```

[How to Run GHCi](#)

`instance Num Int where`
`(+) = addInt`
`(-) = subInt`
`(*) = mulInt`

[Video](#)

2.11

`instance Num Double where`
`(+) = addDbl`
`(-) = subDbl`
`(*) = mulDbl`

[Guessing](#)

[Article](#)

Hierarchy of numeric classes

2.12 There are some operations (addition) that are valid for all numeric types.

- There are some others (e.g. trigonometric functions) that are valid only for some numeric types.

[What do you know about Haskell?](#)

- Therefore there is a rich hierarchy of subclasses, including
 - *Integral* — class of numeric types that represent integer values, including *Int*, *Integer*, and more.

2.13

- *Fractional* — class of types that can represent fractions.

- *Floating* — class containing *Float*, *Double*, etc.

- *Bounded* — class of numeric types that have a minimal and maximal element.

[Video](#)

- *Bits* — class of types where you can access the representation as a sequence of bits, useful for systems programming and digital circuit design.

Week 3:

- If you want to get deeply into numeric classes and types, refer to [the Haskell documentation](#).

Data Structures and Types

Functions on Lists

The Show class

3.1 • We have been using *show* to convert a data value to a string, which can then be written to output.

[Welcome to Week 3](#)

- Some values can be “shown”, but not all.

- For example, it is impossible in general to show a function.

- Therefore *show* needs a type class!

- $show :: Show\ a \Rightarrow a \rightarrow String$

3.2

Defining your own Show instance

[Recursive Functions on Lists](#)

[Article](#)

```
data Foo = Bar | Baz
```

3.3

We might want our own peculiar string representation:

[Functional Maps and Folds versus Imperative Loops](#)

```
instance Show Foo where
```

```
show Bar = "it is a bar"
```

```
show Baz = "this is a baz"
```

3.4

Recall that when you enter an expression *exp* into *ghci*, it prints *showexp*. So we can

[Try out our strange instance declaration:](#)

[Exercise](#)

```
*Main> Bar
```

```
it is a bar
```

3.5

```
*Main> Baz
```

```
this is a baz
```

[Do it Yourself: Function Composition](#)

Deriving Show

[Exercise](#)

This is a similar type, but it has a *deriving* clause.

3.6

[What Have We Learned About Lists?](#)

```
data Foo2 = Bar2 | Baz2
```

```
deriving (Read, Show)
```

[Quiz](#)

Haskell will automatically define an instance of *show* for *Foo2*, using the obvious definition:

Write a Spelling Book Generator

```
*Main> Bar2
Bar2
Exercise
*Main> Baz2
Baz2
```

3.8

More standard typeclasses

Summary

Here is a summary of some of the type classes defined in the standard libraries.

Article

- *Num* — numbers, with many subclasses for specific kinds of number.
- *Read* — types that can be “read in from” a string.
- *Show* — types that can be “shown to” a string.
- *Eq* — types for which the equality operator `==` is defined.
- *Ord* — types for which you can do comparisons like `<`, `>`, etc.
- *Enum* — types where the values can be enumerated in sequence; this is used for example in the notation `[1..n]` and `'a'..'z'`.

3.9

Define Your Own Data Types

Video

```
*Main> [1..10]
3.1([1,2,3,4,5,6,7,8,9,10]
*Main> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

Grow a Tree

© University of Glasgow

Article

Share this article:

Type Classes

Video

Haskell History
2 comments

Previous

Mark as complete
Interview with Simon Peyton Jones

Next

Mark as complete

Video

Comments

Brief History of Haskell

Article P.

Add a comment...
(plain text and
markdown available)

Article

0/1200

3 15
Post [Learn more about markdown](#)

Show:
End of Week 3

All comments ▼

Sort by:

Newest ▾

Filter

Week 4

When Programs Get Bigger

[Gwee YC](#)

Program Structure

Follow

[31 OCT](#) [31 OCT](#)

4.1 I think it can be coded in ways more than an interface can, but to me, it seems basically like overloading operators to deal with a set of meaningful objects the programmer wants to lump together, where the various 'domains' of the operations are all abstracted as the 'one domain' of the type class.

[Video](#)

(edited)

4.2

Like

[Reply](#)[Keep your programs Tidy](#)

Bookmark

[Article](#) [Report](#)

4.3

[Xavier Góngora](#)[Guard your shards!](#)

Follow

[Article](#) [20 OCT](#) [20 OCT](#)

4.4 So a type class can be described equivalently as a:

[Dealing with Uncertainty](#) 1. Set of types (instantiated) with an implementation of a family of functions that describe the behavior of the class.

[Video](#) 2. An interface to restrict a family of polymorphic function to a set of types.

4.5 Is this accurate? I'm trying to have my computer science concepts right.

[Idiom](#) [Like](#) [Haskell](#)[Reply](#)[Quiz](#)

Bookmark

[Report](#)

Parsing Text

- [Help Centre](#)
- [Child safety](#)

4.6 • [Privacy](#)

- [T&Cs](#)

[Parsing Text Using Higher-Order Functions](#)[Contact FutureLearn for Support](#)[Article](#)

Our website is updated regularly so this content may now be out of date, please go to <https://www.futurelearn.com> for the most up to date information.

[Parsing using Parsec: a practical example](#)[Video](#)

4.8

[Parser Puzzles](#)

[Quiz](#)

4.9

[Summary](#)

[Article](#)

Am I Right?

4.10

[Check my Program is Correct](#)

[Video](#)

4.11

[Using QuickCheck](#)

[Article](#)

4.12

[Talk with a Haskell Teacher](#)

[Video](#)

Week 5

Hardcore Haskell

Laziness and Infinite Data structures

5.1

[Welcome to Week 5](#)

[Video](#)

5.2

[Lazy is Good](#)

[Video](#)

5.3

[Infinite Data Structures](#)

[Article](#)

5.4

[To Infinity \(but not beyond\)](#)

[Quiz](#)

More about Types

5.5

[Type Horror Stories](#)

[Discussion](#)

5.6

[Types, lambda functions and type classes](#)

[Article](#)

5.7

[Curry is on the menu](#)

[Video](#)

5.8

[Type Inference by Example](#)

[Video](#)

5.9

[You are the type checker](#)

[Quiz](#)

5.10

[Summary](#)

[Article](#)

Haskell in the Real World

5.11

[Haskell at Facebook](#)

[Video](#)

5.12

[Haskell in the Wild](#)[Article](#)

5.13

[Course Feedback](#)[Article](#)

Week 6

Think like a Functional Programmer

Type Classes

6.1

[Welcome to Week 6](#)[Video](#)

6.2

[Types with Class](#)[Video](#)

6.3

[Type Classes in more Detail](#)[Article](#)

6.4

[Summary](#)[Article](#)

Geek Greek

6.5

[Introduction to the Lambda calculus](#)[Article](#)

6.6

[There are Only Functions! \(Optional\)](#)[Video](#)

6.7

[We Love Lambda!](#)

[Quiz](#)

6.8

[Summary](#)

[Article](#)

The M-word

6.9

[We Already Know About Monads](#)

[Video](#)

6.10

[Introduction to monad theory](#)

[Article](#)

6.11

[Example: the Maybe monad](#)

[Article](#)

6.12

[Monad metaphors](#)

[Discussion](#)

6.13

[Summary](#)

[Article](#)

So long and thanks for all the fun(ctions)!

6.14

[Functional Programming in Other Languages](#)

[Video](#)

6.15

[Will You Use Haskell in the Future?](#)

[Discussion](#)

6.16

[The End of the Affair](#)

[Video](#)