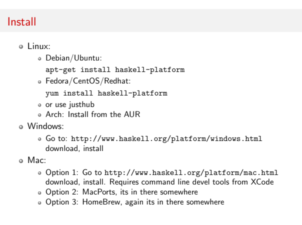
2013-05-25

Intro to Haskell
└─Getting Started
  └─Install
    └─Install



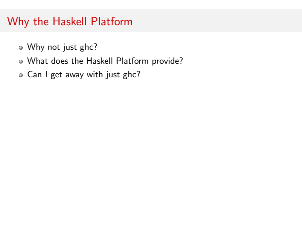
Install

- Linux:
  - Debian/Ubuntu:
    `apt-get install haskell-platform`
  - Fedora/CentOS/Redhat:
    `yum install haskell-platform`
  - or use justhub
  - Arch: Install from the AUR
- Windows:
  - Go to: `http://www.haskell.org/platform/windows.html` download, install
- Mac:
  - Option 1: Go to `http://www.haskell.org/platform/mac.html` download, install. Requires command line devel tools from XCode
  - Option 2: MacPorts, its in there somewhere
  - Option 3: HomeBrew, again its in there somewhere

1. Arch users are entirely capable of breaking their system on their own, and should know how to use the AUR
2. People who know about macports and homebrew should be good enough to get this it working

2013-05-25

Intro to Haskell
└─Getting Started
  └─Why the Haskell Platform
    └─Why the Haskell Platform



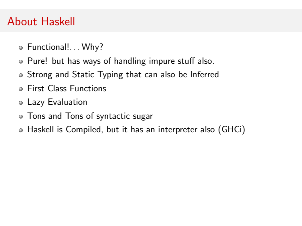
Why the Haskell Platform

- Why not just ghc?
- What does the Haskell Platform provide?
- Can I get away with just ghc?

1. The haskell platform provides a large set of libraries that work well together so you can get started quicker and easier. Think of it like Python's 'Batteries', but for haskell
2. Yes, but it will suck, just go to Alamode to find out how much it sucks to not have basic libraries

2013-05-25

Intro to Haskell
└Basics
  └About
    └About Haskell



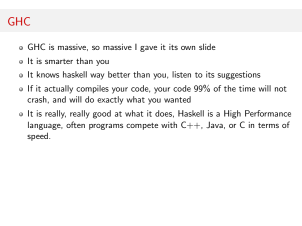
About Haskell

- Functional!. . . Why?
- Pure! but has ways of handling impure stuff also.
- Strong and Static Typing that can also be Inferred
- First Class Functions
- Lazy Evaluation
- Tons and Tons of syntactic sugar
- Haskell is Compiled, but it has an interpreter also (GHCi)

1. Its different, fun, and not always practical, but it can easily solve problems that imperative languages can't
2. No side-effects you don't have to worry about that Butterfly flipping a bit while you're here: reference: http://xkcd.com/378/
3. Purity also implies that everything will be immutable
4. Think C/C++ but without nice implicit type casting But you don't have to tell haskell you are using a String or Int or whatever if you don't want to, ghc will figure this out
5. A function is no different than a value, you are encouraged (and need) to make functions that take functions as parameters, and return functions themselves
6. You won't directly deal with this, but haskell doesn't actually compute anything until you ask for it.
7. Learn to love operators and symbols, you will use them a lot
8. GHCi is where everyone should start, learning haskell

2013-05-25

Intro to Haskell
└Basics
  └GHC
    └GHC

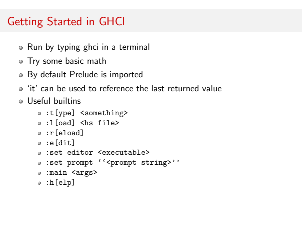


GHC

- GHC is massive, so massive I gave it its own slide
- It is smarter than you
- It knows haskell way better than you, listen to its suggestions
- If it actually compiles your code, your code 99% of the time will not crash, and will do exactly what you wanted
- It is really, really good at what it does, Haskell is a High Performance language, often programs compete with C++, Java, or C in terms of speed.

1. Its like a 700MB executable
2. Don't argue with it, you will lose
3. It will actually give you suggestions on how to fix common problems or bad ideas
4. No Joke

Intro to Haskell
└─Coding Environment
    └─GHCI
        └─Getting Started in GHCI

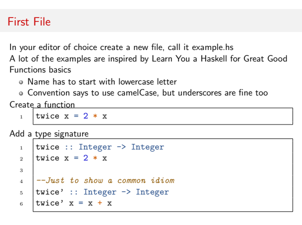


Getting Started in GHCI

- Run by typing ghci in a terminal
- Try some basic math
- By default Prelude is imported
- ‘it’ can be used to reference the last returned value
- Useful builtins
    - `:t[ype] <something>`
    - `:l[oad] <hs file>`
    - `:r[eload]`
    - `:e[dit]`
    - `:set editor <executable>`
    - `:set prompt ‘‘<prompt string>’’`
    - `:main <args>`
    - `:h[elp]`

1. Prelude is the most basic default functions that you will probably want and need for any project, there are other Preludes also
2. Gives you the type definition of something
3. Loads a haskell file into ghci so you can run and test code
4. Reloads last loaded file
5. Opens the last loaded file in your editor
6. Sets you editor to be whatever you tell it
7. Sets your prompt, by law it should be a λ or some other haskell-ly thing
8. Runs the main function of the loaded file, with args, simulating running it from your shell like normal
9. Displays GHCi help, telling you all about these commands and more

Intro to Haskell
└─Coding Environment
    └─File
        └─First File



First File

In your editor of choice create a new file, call it example.hs
A lot of the examples are inspired by Learn You a Haskell for Great Good
Functions basics

- Name has to start with lowercase letter
- Convention says to use camelCase, but underscores are fine too

Create a function

```
twice x = 2 * x
```

Add a type signature

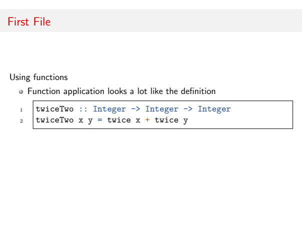
```
twice :: Integer -> Integer
twice x = 2 * x

--Just to show a common idiom
twice' :: Integer -> Integer
twice' x = x + x
```

1. Uppercase names are reserved for Type Constructors
2. Explain how the type signature works, we will get more into detail later

2013-05-25

Intro to Haskell
└─Coding Environment
  └─File
    └─First File



First File

Using functions

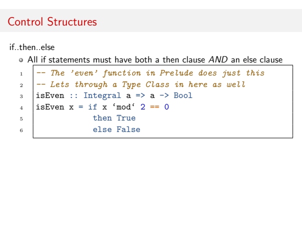
- Function application looks a lot like the definition

```
twiceTwo :: Integer -> Integer -> Integer
twiceTwo x y = twice x + twice y
```

1. Explain how the type signature works, we will get more into detail later
2. Have them load their code into GHCi, and try out their functions

2013-05-25

Intro to Haskell
└─The Basics
  └─Control
    └─Control Structures



Control Structures

if..then..else

- All if statements must have both a then clause *AND* an else clause

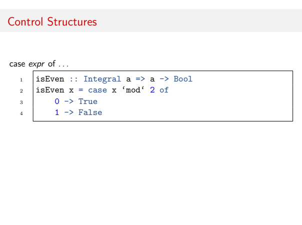
```
-- The 'even' function in Prelude does just this
-- Lets through a Type Class in here as well
isEven :: Integral a => a -> Bool
isEven x = if x `mod` 2 == 0
             then True
             else False
```

Make sure they understand this is an expression not a statement

1. Welcome to Type Classes, broad overview here
2. Integral is a Type of Real and Enum, and contains Int and Integer
3. Relate if to the ternary operator

2013-05-25

Intro to Haskell
└─The Basics
  └─Control
    └─Control Structures



Control Structures

case *expr* of . . .

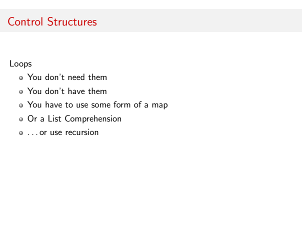
```
isEven :: Integral a => a -> Bool
isEven x = case x `mod` 2 of
    0 -> True
    1 -> False
```

Make sure they understand this is an expression not a statement

1. Welcome to Type Classes, broad overview here
2. Integral is a Type of Real and Enum, and contains Int and Integer
3. Relate if to the ternary operator

2013-05-25

Intro to Haskell
└─The Basics
  └─Control
    └─Control Structures



Control Structures

Loops

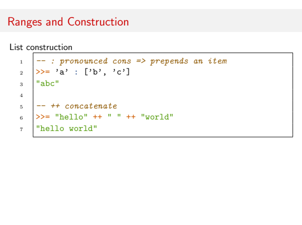
- You don't need them
- You don't have them
- You have to use some form of a map
- Or a List Comprehension
- . . . or use recursion

Make sure they understand this is an expression not a statement

1. Welcome to Type Classes, broad overview here
2. Integral is a Type of Real and Enum, and contains Int and Integer
3. Relate if to the ternary operator
4. In 2 frames I cover maps, folds, filters, and list comps
5. Next frame is lists

2013-05-25

Intro to Haskell
└─Lists
└─Building
└─Ranges and Construction



Ranges and Construction

List construction

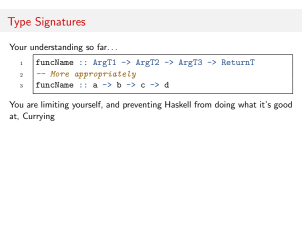
```
-- : pronounced cons => prepends an item
>>= 'a' : ['b', 'c']
"abc"

-- ++ concatenate
>>= "hello" ++ " " ++ "world"
"hello world"
```

1. Strings are really just [Char]

2013-05-25

Intro to Haskell
└─The Type System
└─Type Signatures
└─Type Signatures



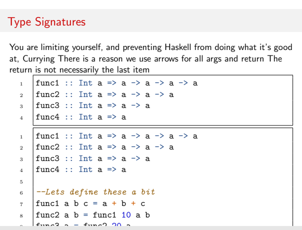
Type Signatures

Your understanding so far...

```
funcName :: ArgT1 -> ArgT2 -> ArgT3 -> ReturnT
-- More appropriately
funcName :: a -> b -> c -> d
```

You are limiting yourself, and preventing Haskell from doing what it's good at, Currying

well actually it is what is on the rightNow is a really good time to talk about lambda calculusLambda calculus' functions always take 1 value, but can return functionsMake them ask questions, make sure they understand



Type Signatures

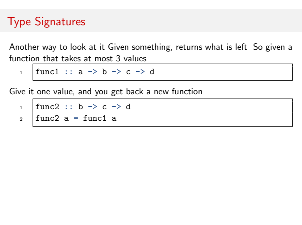
You are limiting yourself, and preventing Haskell from doing what it's good at, Currying There is a reason we use arrows for all args and return The return is not necessarily the last item

```haskell
func1 :: Int a => a -> a -> a -> a
func2 :: Int a => a -> a -> a
func3 :: Int a => a -> a
func4 :: Int a => a
```

```haskell
func1 :: Int a => a -> a -> a -> a
func2 :: Int a => a -> a -> a
func3 :: Int a => a -> a
func4 :: Int a => a

--Lets define these a bit
func1 a b c = a + b + c
func2 a b = func1 10 a b
func3 a = func2 20 a
```

well actually it is what is on the rightNow is a really good time to talk about lambda calculusLambda calculus' functions always take 1 value, but can return functionsMake them ask questions, make sure they understand



Type Signatures

Another way to look at it Given something, returns what is left So given a function that takes at most 3 values

```haskell
func1 :: a -> b -> c -> d
```

Give it one value, and you get back a new function

```haskell
func2 :: b -> c -> d
func2 a = func1 a
```

well actually it is what is on the rightNow is a really good time to talk about lambda calculusLambda calculus' functions always take 1 value, but can return functionsMake them ask questions, make sure they understand