Table of Contents

IO

Benson Joeris
benson@bjoeris.com
http://bjoeris.com

pluralsight
hardcore developer training

Overview

- Compile executable program
- IO Actions
- Combine IO Actions
- IO Values
- Useful IO Functions

Table of Contents

Hello World

```
main = putStrLn "Hello World"
```

1. Save as ``HelloWorld.hs''
2. Open up a terminal in that folder
3. Type ``ghc HelloWorld.hs''
4. Run HelloWorld executable

Hello World

```
main = putStrLn "Hello World"
```

- Haskell functions are pure
  - Cannot modify any external state
  - Value cannot depend on external state
  - Cannot write to the console

Table of Contents

IO Actions

```
main = putStrLn "Hello World"
```

```
putStrLn :: String -> IO ()
```

- ()

```
data Unit = Unit
```

IO Actions

```
main = putStrLn "Hello World"
```

```
putStrLn :: String -> IO ()
```

- IO

```
data IO a
```

```
data Maybe a
```

IO Actions

```
main = putStrLn "Hello World"
```

```
putStrLn :: String -> IO ()
```

IO Actions

```
main :: IO ()
main = putStrLn "Hello World"
```

```
putStrLn :: String -> IO ()
```

- `main` –IO action executed by the program

IO Actions

```
main :: IO ()
main = putStrLn "Hello World" -- printed

main2 :: IO ()
main2 = putStrLn "Hello World 2" -- not printed
```

Table of Contents

do-Blocks

```
main :: IO ()
main = do
  putStrLn "Hello"
  putStrLn "World"
```

Output:

```
Hello
World
```

do-Blocks

```
main :: IO ()
main = do
  putStrLn "Hello"
  putStrLn "World"
x = 3 -- no longer part of do-block
```

Output:

```
Hello
World
```

do-Blocks

```
helloWorld :: IO ()
helloWorld = putStrLn "Hello World"

main :: IO ()
main = do
  helloWorld
  helloWorld
  helloWorld
```

Output:

```
Hello World
Hello World
Hello World
```

do-Blocks

```
introduce :: String -> String -> IO ()
introduce name1 name2 = do
  putStrLn (name1 ++ ", this is " ++ name2)
  putStrLn (name2 ++ ", this is " ++ name1)

main :: IO ()
main = do
  introduce "Alice" "Bob"
  introduce "Alice" "Sally"
```

Output:

```
Alice, this is Bob
Bob, this is Alice
Alice, this is Sally
Sally, this is Alice
```

Table of Contents

IO Values

```
main :: IO ()
main = do
  line <- getLine
  putStrLn ("You said: " ++ line)
```

Input:

```
blah blah blah
```

Output:

```
You said: blah blah blah
```

IO Values

```
main :: IO ()
main = do
  line <- getLine
  putStrLn ("You said: " ++ line)
```

Input:

```
blah blah blah
oh look, another line
```

Output:

```
You said: blah blah blah
```

IO Values

```
main :: IO ()
main = do
  line <- getLine
  putStrLn ("You said: " ++ line)
```

```
getLine :: IO String
```

- <– *ONLY* inside `do`-block
- Bound variable can only be used later in the same `do`-block

IO Values

```
greet :: IO ()
greet = do
  putStrLn "Who are you?"
  who <- getLine
  putStrLn ("Hello " ++ who)

greetForever :: IO ()
greetForever = do
  greet
  greetForever

main :: IO ()
main = greetForever
```

IO Values

```
extractValue :: IO a -> a
```

Table of Contents

`return` Function

```
dummyGetLine :: IO String
dummyGetLine =
  return "I'm not really doing anything"

main :: IO ()
main = do
  line <- dummyGetLine
  putStrLn line
```

```
return :: a -> IO a
```

`return` Function

```
promptInfo :: IO (String,String)
promptInfo = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn "What is your favorite color?"
  color <- getLine
  return (name,color)

main :: IO ()
main = do
  (name,color) <- promptInfo
  putStrLn ("Hello " ++ name)
  putStrLn ("I like " ++ color ++ " too!")
```

`return` Function

```
promptInfo :: IO (String,String)
promptInfo = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn "What is your favorite color?"
  color <- getLine
  (name,color)

main :: IO ()
main = do
  (name,color) <- promptInfo
  putStrLn ("Hello " ++ name)
  putStrLn ("I like " ++ color ++ " too!")
```

`return` Function

```
promptInfo :: IO (String,String)
promptInfo = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn "What is your favorite color?"
  color <- getLine
  return (name,color)

main :: IO ()
main = do
  (name,color) <- promptInfo
  putStrLn ("Hello " ++ name)
  putStrLn ("I like " ++ color ++ " too!")
```

`return` Function

```
main :: IO ()
main = do
  line1 <- getLine
  line2 <- getLine
  putStrLn (line1 ++ line2)
```

`return` Function

```
main :: IO ()
main = do
  line1 <- getLine
  line2 <- getLine
  lines <- line1 ++ line2
  putStrLn lines
```

-

`return` Function

```
main :: IO ()
main = do
  line1 <- getLine
  line2 <- getLine
  lines <- return (line1 ++ line2)
  putStrLn lines
```

`return` Function

```
main :: IO ()
main = do
  line1 <- getLine
  line2 <- getLine
  let lines = line1 ++ line2
  putStrLn lines
```

`return` Function

```
main :: IO ()
main = do
  return 0
  putStrLn "haha, still running"
  return "halt!"
  putStrLn "you can't stop me!"
```

```
return :: a -> IO a
```

Output:

```
haha, still running
you can't stop me!
```

Table of Contents

Some Useful IO Actions

```
putStrLn :: String -> IO ()
```

- Print a string to the console, and append a new line

```
getLine :: IO String
```

- Reads a line from the console

```
print :: (Show a) => a -> IO ()
```

- Print string representation of a value -

Some Useful IO Actions

```
readFile :: FilePath -> IO String
```

- Read an entire file as a (lazy) string

```
writeFile :: FilePath -> String -> IO ()
```

- Write a string to a file

```
appendFile :: FilePath -> String -> IO ()
```

- Appends a string to a file

```
type FilePath = String
```

Some Useful IO Actions

```
interact :: (String -> String) -> IO ()
```

```
reverseLines :: String -> String
reverseLines input =
  unlines (map reverse (lines input))

main :: IO ()
main = interact reverseLines
```

Table of Contents

Program Organization

- Do as little in IO as possible

Program Organization

$$A \mapsto B$$

$$B \mapsto C$$

$$C \mapsto D$$

$$\vdots$$

$$Z \mapsto A$$

```
encrypt :: Char -> Char
encrypt c
  | 'A' <= c && c < 'Z' =
      toEnum (fromEnum 'A' + 1)
  | c == 'Z' = 'A'
  | otherwise = c
```

Program Organization

```
encrypt :: Char -> Char
encrypt = ···

handleChar :: IO ()
handleChar = do
  c <- getChar
  let u = encrypt c
  putChar c

inputLoop :: IO ()
inputLoop = do
  handleChar
  inputLoop

main :: IO ()
main = inputLoop
```

Program Organization

```
encrypt :: Char -> Char
encrypt = ···

main :: IO ()
main = interact (map encrypt)
```

Table of Contents

Summary

- Compiling
- IO Actions
- IO Values
- *return*
- *do*-Blocks
- Some Useful IO Actions
- Program Organization