# Haskell in the real World: The IO monad

Christoph “Hammy“ Stahl

February 25, 2021

# Überblick

# Software

Software:

- Linux
- GHC
- cabal
- Download: `http://hackage.haskell.org/platform/`
- Oder aus dem eigenen Paketmanager.

# Software

Software:

- Linux
- GHC
- cabal
- Download: `http://hackage.haskell.org/platform/`
- Oder aus dem eigenen Paketmanager.

# Software

Software:

- Linux
- GHC
- cabal
- Download: `http://hackage.haskell.org/platform/`
- Oder aus dem eigenen Paketmanager.

# Software

Software:

- Linux
- GHC
- cabal
- Download: http://hackage.haskell.org/platform/
- Oder aus dem eigenen Paketmanager.

# Software

Software:

- Linux
- GHC
- cabal
- Download: http://hackage.haskell.org/platform/
- Oder aus dem eigenen Paketmanager.

# cabal

- Cabal ist ein Paketmanager für Haskell
- Benutzt, um Bibliotheken und Programme zu installieren, und dabei automatisch Abhängigkeiten aufzulösen,
- Teilweise müssen entsprechende C Bibliotheken installiert sein.
- Läd die Programme und Abhängigkeiten aus dem Hackage.

Shell:

```
$ cabal install <paketname>
```

# cabal

- Cabal ist ein Paketmanager für Haskell
- Benutzt, um Bibliotheken und Programme zu installieren, und dabei automatisch Abhängigkeiten aufzulösen,
- Teilweise müssen entsprechende C Bibliotheken installiert sein.
- Läd die Programme und Abhängigkeiten aus dem Hackage.

Shell:

```
$ cabal install <paketname>
```

# cabal

- Cabal ist ein Paketmanager für Haskell
- Benutzt, um Bibliotheken und Programme zu installieren, und dabei automatisch Abhängigkeiten aufzulösen,
- Teilweise müssen entsprechende C Bibliotheken installiert sein.
- Läd die Programme und Abhängigkeiten aus dem Hackage.

Shell:

```
$ cabal install <paketname>
```

# cabal

- Cabal ist ein Paketmanager für Haskell
- Benutzt, um Bibliotheken und Programme zu installieren, und dabei automatisch Abhängigkeiten aufzulösen,
- Teilweise müssen entsprechende C Bibliotheken installiert sein.
- Läd die Programme und Abhängigkeiten aus dem Hackage.

Shell:

```
$ cabal install <paketname>
```

# cabal

- Cabal ist ein Paketmanager für Haskell
- Benutzt, um Bibliotheken und Programme zu installieren, und dabei automatisch Abhängigkeiten aufzulösen,
- Teilweise müssen entsprechende C Bibliotheken installiert sein.
- Läd die Programme und Abhängigkeiten aus dem Hackage.

**Shell:**

```
$ cabal install <paketname>
```

# Hackage

- Sammlung von Haskell Bibliotheken und Programmen.
- Online Zugriff auf die Dokumentation der Bibliotheken.
- Vergleichbar mit CPAN oder RubyGems.

# Hackage

- Sammlung von Haskell Bibliotheken und Programmen.
- Online Zugriff auf die Dokumentation der Bibliotheken.
- Vergleichbar mit CPAN oder RubyGems.

# Hackage

- Sammlung von Haskell Bibliotheken und Programmen.
- Online Zugriff auf die Dokumentation der Bibliotheken.
- Vergleichbar mit CPAN oder RubyGems.

# Was ist die IO Monade

- Kapselt IO Zugriff.
- Kommunikation mit der "Realen" Welt.
- Möglichkeit das Programm zu steuern.
- Potential für Seiteneffekte.

# Was ist die IO Monade

- Kapselt IO Zugriff.
- Kommunikation mit der "Realen" Welt.
- Möglichkeit das Programm zu steuern.
- Potential für Seiteneffekte.

# Was ist die IO Monade

- Kapselt IO Zugriff.
- Kommunikation mit der "Realen" Welt.
- Möglichkeit das Programm zu steuern.
- Potential für Seiteneffekte.

# Was ist die IO Monade

- Kapselt IO Zugriff.
- Kommunikation mit der "Realen" Welt.
- Möglichkeit das Programm zu steuern.
- Potential für Seiteneffekte.

# main-Funktion

- Startpunkt eines jeden Haskell Programmes.

Haskell:

```
main :: IO ()
```

- Muss demnach ein **IO** () zurück geben.

Beispiel:

```
main = do
  foo <- barIOfunction "text"
  let stuff = map (pureFunction) foo
  putStrLn stuff -- putStrLn :: IO ()
```

# main-Funktion

- Startpunkt eines jeden Haskell Programmes.

**Haskell:**

```
main :: IO ()
```

- Muss demnach ein **IO ()** zurück geben.

**Beispiel:**

```
main = do
  foo <- barIOfunction "text"
  let stuff = map (pureFunction) foo
  putStrLn stuff -- putStrLn :: IO ()
```

# main-Funktion

- Startpunkt eines jeden Haskell Programmes.

**Haskell:**

```
main :: IO ()
```

- Muss demnach ein **IO** () zurück geben.

**Beispiel:**

```
main = do
  foo <- barIOfunction "text"
  let stuff = map (pureFunction) foo
  putStrLn stuff -- putStrLn :: IO ()
```

# main-Funktion

- Startpunkt eines jeden Haskell Programmes.

**Haskell:**

```
main :: IO ()
```

- Muss demnach ein **IO** () zurück geben.

**Beispiel:**

```
main = do
  foo <- barIOfunction "text"
  let stuff = map (pureFunction) foo
  putStrLn stuff -- putStrLn :: IO ()
```

# stdin/stdout

- Jedes Programm hat 3 standard Datenströme

  stdin Standardeingabe, meist Tastatur.
  stdout Standardausgabe, meist Terminal.
  stderr Standardausgabe für Fehler, meist Terminal.

- Können in Dateien oder an andere Programme gelenkt werden.

Shell:

```
$ prog1 < file | prog2 > file 2> error.log
```

# stdin/stdout

- Jedes Programm hat 3 standard Datenströme
  - stdin Standardeingabe, meist Tastatur.
  - stdout Standardausgabe, meist Terminal.
  - stderr Standardausgabe für Fehler, meist Terminal.
- Können in Dateien oder an andere Programme gelenkt werden.

Shell:

```
$ prog1 < file | prog2 > file 2> error.log
```

# stdin/stdout

- Jedes Programm hat 3 standard Datenströme
  - stdin Standardeingabe, meist Tastatur.
  - stdout Standardausgabe, meist Terminal.
  - stderr Standardausgabe für Fehler, meist Terminal.
- Können in Dateien oder an andere Programme gelenkt werden.

Shell:

```
$ prog1 < file | prog2 > file 2> error.log
```

# stdin/stdout

- Jedes Programm hat 3 standard Datenströme
  - stdin Standardeingabe, meist Tastatur.
  - stdout Standardausgabe, meist Terminal.
  - stderr Standardausgabe für Fehler, meist Terminal.
- Können in Dateien oder an andere Programme gelenkt werden.

Shell:

```
$ prog1 < file | prog2 > file 2> error.log
```

# stdin/stdout

- Jedes Programm hat 3 standard Datenströme

  stdin Standardeingabe, meist Tastatur.
  stdout Standardausgabe, meist Terminal.
  stderr Standardausgabe für Fehler, meist Terminal.

- Können in Dateien oder an andere Programme gelenkt werden.

Shell:

```
$ prog1 < file | prog2 > file 2> error.log
```

# stdin/stdout

- Jedes Programm hat 3 standard Datenströme
  - stdin Standardeingabe, meist Tastatur.
  - stdout Standardausgabe, meist Terminal.
  - stderr Standardausgabe für Fehler, meist Terminal.
- Können in Dateien oder an andere Programme gelenkt werden.

**Shell:**

```
$ prog1 < file | prog2 > file 2> error.log
```

# stdout

Wichtige Funktionen:

**putStr** Schreibt einen String in die Ausgabe.

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

**putStrLn** Schreibt einen String in die Ausgabe, gefolgt von einem Newline.

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

**putChar** Schreibt einen Buchstaben in die Ausgabe.

```
putChar ::  Char -> IO ()
```

**print** Schreibt eine Variable in die Ausgabe, gefolgt von einem Newline.

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

# stdout

Wichtige Funktionen:

**putStr** Schreibt einen String in die Ausgabe.

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

**putStrLn** Schreibt einen String in die Ausgabe, gefolgt von einem Newline.

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

**putChar** Schreibt einen Buchstaben in die Ausgabe.

```
putChar ::  Char -> IO ()
```

**print** Schreibt eine Variable in die Ausgabe, gefolgt von einem Newline.

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

# stdout

Wichtige Funktionen:

**putStr** Schreibt einen String in die Ausgabe.

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

**putStrLn** Schreibt einen String in die Ausgabe, gefolgt von einem Newline.

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

**putChar** Schreibt einen Buchstaben in die Ausgabe.

```
putChar ::  Char -> IO ()
```

**print** Schreibt eine Variable in die Ausgabe, gefolgt von einem Newline.

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

# stdout

Wichtige Funktionen:

**putStr** Schreibt einen String in die Ausgabe.

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

**putStrLn** Schreibt einen String in die Ausgabe, gefolgt von einem Newline.

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

**putChar** Schreibt einen Buchstaben in die Ausgabe.

```
putChar ::  Char -> IO ()
```

**print** Schreibt eine Variable in die Ausgabe, gefolgt von einem Newline.

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

# Beispiel: Hello World!

Haskell: HelloWorld1.hs

```
main = do
  putStrLn "Hello "
  putStrLn "World!"
```

Ausgabe:
```
Hello
World!
```

Haskell: HelloWorld2.hs

```
main = do
  putStr "Hello "
  putStr "World!"
```

Ausgabe:
```
Hello World!
```

Haskell: Umstaendlich.hs

```
main = do
  putChar 'C'
  putChar 'h'
  putChar 'a'
  putChar 'r'
  putChar '!'
  putChar '\n'
```

Ausgabe:
```
Char!
```

Haskell: Add.hs

```
main = do
  print $ 3+4
```

Ausgabe:
```
7
```

# Beispiel: Hello World!

**Haskell: HelloWorld1.hs**

```
main = do
  putStrLn "Hello "
  putStrLn "World!"
```

Ausgabe
```
Hello
World!
```

**Haskell: HelloWorld2.hs**

```
main = do
  putStr "Hello "
  putStr "World!"
```

Ausgabe Hello World!

**Haskell: Umstaendlich.hs**

```
main = do
  putChar 'C'
  putChar 'h'
  putChar 'a'
  putChar 'r'
  putChar '!'
  putChar '\n'
```

Ausgabe Char!

**Haskell: Add.hs**

```
main = do
  print $ 3+4
```

Ausgabe 7

# Beispiel: Hello World!

**Haskell: HelloWorld1.hs**

```
main = do
  putStrLn "Hello "
  putStrLn "World!"
```

Ausgabe
```
Hello
World!
```

**Haskell: HelloWorld2.hs**

```
main = do
  putStr "Hello "
  putStr "World!"
```

Ausgabe Hello World!

**Haskell: Umstaendlich.hs**

```
main = do
  putChar 'C'
  putChar 'h'
  putChar 'a'
  putChar 'r'
  putChar '!'
  putChar '\n'
```

Ausgabe Char!

**Haskell: Add.hs**

```
main = do
  print $ 3+4
```

Ausgabe 7

# Beispiel: Hello World!

**Haskell: HelloWorld1.hs**

```
main = do
  putStrLn "Hello "
  putStrLn "World!"
```

Ausgabe
```
Hello
World!
```

**Haskell: HelloWorld2.hs**

```
main = do
  putStr "Hello "
  putStr "World!"
```

Ausgabe `Hello World!`

**Haskell: Umstaendlich.hs**

```
main = do
  putChar 'C'
  putChar 'h'
  putChar 'a'
  putChar 'r'
  putChar '!'
  putChar '\n'
```

Ausgabe `Char!`

**Haskell: Add.hs**

```
main = do
  print $ 3+4
```

Ausgabe `7`

# Beispiel: Hello World!

**Haskell: HelloWorld1.hs**

```
main = do
  putStrLn "Hello "
  putStrLn "World!"
```

Ausgabe
```
Hello
World!
```

**Haskell: HelloWorld2.hs**

```
main = do
  putStr "Hello "
  putStr "World!"
```

Ausgabe `Hello World!`

**Haskell: Umstaendlich.hs**

```
main = do
  putChar 'C'
  putChar 'h'
  putChar 'a'
  putChar 'r'
  putChar '!'
  putChar '\n'
```

Ausgabe `Char!`

**Haskell: Add.hs**

```
main = do
  print $ 3+4
```

Ausgabe `7`

# Beispiel: Hello World!

## Haskell: HelloWorld1.hs

```
main = do
  putStrLn "Hello␣"
  putStrLn "World!"
```

Ausgabe
```
Hello
World!
```

## Haskell: HelloWorld2.hs

```
main = do
  putStr "Hello␣"
  putStr "World!"
```

Ausgabe
```
Hello World!
```

## Haskell: Umstaendlich.hs

```
main = do
  putChar 'C'
  putChar 'h'
  putChar 'a'
  putChar 'r'
  putChar '!'
  putChar '\n'
```

Ausgabe
```
Char!
```

## Haskell: Add.hs

```
main = do
  print $ 3+4
```

Ausgabe
```
7
```

# Beispiel: Hello World!

**Haskell: HelloWorld1.hs**

```
main = do
  putStrLn "Hello "
  putStrLn "World!"
```

Ausgabe
```
Hello
World!
```

**Haskell: HelloWorld2.hs**

```
main = do
  putStr "Hello "
  putStr "World!"
```

Ausgabe
```
Hello World!
```

**Haskell: Umstaendlich.hs**

```
main = do
  putChar 'C'
  putChar 'h'
  putChar 'a'
  putChar 'r'
  putChar '!'
  putChar '\n'
```

Ausgabe
```
Char!
```

**Haskell: Add.hs**

```
main = do
  print $ 3+4
```

Ausgabe
```
7
```

# Beispiel: Hello World!

**Haskell: HelloWorld1.hs**

```
main = do
  putStrLn "Hello "
  putStrLn "World!"
```

Ausgabe
```
Hello
World!
```

**Haskell: HelloWorld2.hs**

```
main = do
  putStr "Hello "
  putStr "World!"
```

Ausgabe
```
Hello World!
```

**Haskell: Umstaendlich.hs**

```
main = do
  putChar 'C'
  putChar 'h'
  putChar 'a'
  putChar 'r'
  putChar '!'
  putChar '\n'
```

Ausgabe
```
Char!
```

**Haskell: Add.hs**

```
main = do
  print $ 3+4
```

Ausgabe
```
7
```

# Beispiel: Hello World!

**Haskell: HelloWorld1.hs**

```
main = do
  putStrLn "Hello "
  putStrLn "World!"
```

Ausgabe
```
Hello
World!
```

**Haskell: HelloWorld2.hs**

```
main = do
  putStr "Hello "
  putStr "World!"
```

Ausgabe
```
Hello World!
```

**Haskell: Umstaendlich.hs**

```
main = do
  putChar 'C'
  putChar 'h'
  putChar 'a'
  putChar 'r'
  putChar '!'
  putChar '\n'
```

Ausgabe
```
Char!
```

**Haskell: Add.hs**

```
main = do
  print $ 3+4
```

Ausgabe
```
7
```

# stdin

Wichtige Funktionen:

**getLine** Liest eine Zeile von der Eingabe.

```
getLine ::  IO String
```

**getChar** Liest ein Zeichen von der Eingabe.

```
getChar ::  IO Char
```

**readLn** Liest eine Zeile von der Eingabe und versucht das Ergebnis zu interpretieren.

```
readLn ::  Read a => IO a
```

Die Funktionen warten, bis Zeichen vorhanden sind.

**getContents** Liest alle Zeichen von der Eingabe. (lazy)

```
getContents ::  IO String
```

Programmfluss geht weiter, bis auf die Eingabe zugegriffen wird.

# stdin

Wichtige Funktionen:

**getLine** Liest eine Zeile von der Eingabe.

```
getLine ::  IO String
```

**getChar** Liest ein Zeichen von der Eingabe.

```
getChar ::  IO Char
```

**readLn** Liest eine Zeile von der Eingabe und versucht das Ergebnis zu interpretieren.

```
readLn ::  Read a => IO a
```

Die Funktionen warten, bis Zeichen vorhanden sind.

**getContents** Liest alle Zeichen von der Eingabe. (lazy)

```
getContents ::  IO String
```

Programmfluss geht weiter, bis auf die Eingabe zugegriffen wird.

# stdin

Wichtige Funktionen:

**getLine** Liest eine Zeile von der Eingabe.

```
getLine ::  IO String
```

**getChar** Liest ein Zeichen von der Eingabe.

```
getChar ::  IO Char
```

**readLn** Liest eine Zeile von der Eingabe und versucht das Ergebnis zu interpretieren.

```
readLn ::  Read a => IO a
```

Die Funktionen warten, bis Zeichen vorhanden sind.

**getContents** Liest alle Zeichen von der Eingabe. (lazy)

```
getContents ::  IO String
```

Programmfluss geht weiter, bis auf die Eingabe zugegriffen wird.

# stdin

Wichtige Funktionen:

**getLine** Liest eine Zeile von der Eingabe.

```
getLine ::  IO String
```

**getChar** Liest ein Zeichen von der Eingabe.

```
getChar ::  IO Char
```

**readLn** Liest eine Zeile von der Eingabe und versucht das Ergebnis zu interpretieren.

```
readLn ::  Read a => IO a
```

Die Funktionen warten, bis Zeichen vorhanden sind.

**getContents** Liest alle Zeichen von der Eingabe. (lazy)

```
getContents ::  IO String
```

Programmfluss geht weiter, bis auf die Eingabe zugegriffen wird.

# stdin

Wichtige Funktionen:

**getLine** Liest eine Zeile von der Eingabe.

```
getLine ::  IO String
```

**getChar** Liest ein Zeichen von der Eingabe.

```
getChar ::  IO Char
```

**readLn** Liest eine Zeile von der Eingabe und versucht das Ergebnis zu interpretieren.

```
readLn ::  Read a => IO a
```

Die Funktionen warten, bis Zeichen vorhanden sind.

**getContents** Liest alle Zeichen von der Eingabe. (lazy)

```
getContents ::  IO String
```

Programmfluss geht weiter, bis auf die Eingabe zugegriffen wird.

# stdin

Wichtige Funktionen:

**getLine** Liest eine Zeile von der Eingabe.

```
getLine ::  IO String
```

**getChar** Liest ein Zeichen von der Eingabe.

```
getChar ::  IO Char
```

**readLn** Liest eine Zeile von der Eingabe und versucht das Ergebnis zu interpretieren.

```
readLn ::  Read a => IO a
```

Die Funktionen warten, bis Zeichen vorhanden sind.

**getContents** Liest alle Zeichen von der Eingabe. (lazy)

```
getContents ::  IO String
```

Programmfluss geht weiter, bis auf die Eingabe zugegriffen wird.

# Beispiel: Hello <User>

## Haskell: HelloUser.hs

```
main = do
  putStrLn "Name?"
  name <- getLine
  putStrLn $ "Hello " ++ name
```

Ausgabe
```
Name?
Hammy
Hello Hammy
```

# Beispiel: Hello <User>

## Haskell: HelloUser.hs

```
main = do
  putStrLn "Name?"
  name <- getLine
  putStrLn $ "Hello " ++ name
```

Ausgabe
```
Name?
Hammy
Hello Hammy
```

# Bibliothek: System.IO

- Standardbibliothek für IO in Haskell
- Prelude exportiert nur einige Funktionen aus System.IO
- Funktionen für Datei Zugriff
- Allgemeiner: Auf Handles zugreifen

# Handles

Mit Handles kann man Datenströme verwalten

- Ein/Ausgabe
- Dateien lesen/schreiben
- In einen Netzwerkstream schreiben/aus einem Netzwerkstream lesen.

# Handles

Mit Handles kann man Datenströme verwalten

- Ein/Ausgabe
- Dateien lesen/schreiben
- In einen Netzwerkstream schreiben/aus einem Netzwerkstream lesen.

# Handles

Mit Handles kann man Datenströme verwalten

- Ein/Ausgabe
- Dateien lesen/schreiben
- In einen Netzwerkstream schreiben/aus einem Netzwerkstream lesen.

# Eigenschaften von Handles

Ein Handle kann

- Eingabe(input), Ausgabe(output) oder Beides verwalten
- offen(open), geschlossen(closed) oder halbgeschlossen(semi-closed) sein

# Eigenschaften von Handles

Ein Handle kann

- Eingabe(input), Ausgabe(output) oder Beides verwalten
- offen(open), geschlossen(closed) oder halbgeschlossen(semi-closed) sein

# Operationen auf Handles

**hGetChar** Liest ein Zeichen von dem Handle

```
hGetChar ::  Handle -> IO Char
```

**hGetLine** Liest einen String von dem Handle (bis zum Newline)

```
hGetLine ::  Handle -> IO String
```

**hGetContents** Liest den Kompletten Handle aus

```
hGetContents ::  Handle -> IO String
```

**hPutChar** Schreibt ein Zeichen in einen Handle

```
hPutChar ::  Handle -> Char -> IO ()
```

**hPutStr** Schreibt einen String in einen Handle

```
hPutStr ::  Handle -> String -> IO ()
```

**hPutStrLn** Schreibt einen String und ein Newline in einen Handle

```
hPutStrLn ::  Handle -> String -> IO ()
```

**hClose** Schließt einen Handle.

```
hClose ::  Handle -> IO ()
```

# Operationen auf Handles

**hGetChar** Liest ein Zeichen von dem Handle

```
hGetChar ::  Handle -> IO Char
```

**hGetLine** Liest einen String von dem Handle (bis zum Newline)

```
hGetLine ::  Handle -> IO String
```

**hGetContents** Liest den Kompletten Handle aus

```
hGetContents ::  Handle -> IO String
```

**hPutChar** Schreibt ein Zeichen in einen Handle

```
hPutChar ::  Handle -> Char -> IO ()
```

**hPutStr** Schreibt einen String in einen Handle

```
hPutStr ::  Handle -> String -> IO ()
```

**hPutStrLn** Schreibt einen String und ein Newline in einen Handle

```
hPutStrLn ::  Handle -> String -> IO ()
```

**hClose** Schließt einen Handle.

```
hClose ::  Handle -> IO ()
```

# Operationen auf Handles

**hGetChar** Liest ein Zeichen von dem Handle

```
hGetChar ::  Handle -> IO Char
```

**hGetLine** Liest einen String von dem Handle (bis zum Newline)

```
hGetLine ::  Handle -> IO String
```

**hGetContents** Liest den Kompletten Handle aus

```
hGetContents ::  Handle -> IO String
```

**hPutChar** Schreibt ein Zeichen in einen Handle

```
hPutChar ::  Handle -> Char -> IO ()
```

**hPutStr** Schreibt einen String in einen Handle

```
hPutStr ::  Handle -> String -> IO ()
```

**hPutStrLn** Schreibt einen String und ein Newline in einen Handle

```
hPutStrLn ::  Handle -> String -> IO ()
```

**hClose** Schließt einen Handle.

```
hClose ::  Handle -> IO ()
```

# Operationen auf Handles

**hGetChar** Liest ein Zeichen von dem Handle

```
hGetChar ::  Handle -> IO Char
```

**hGetLine** Liest einen String von dem Handle (bis zum Newline)

```
hGetLine ::  Handle -> IO String
```

**hGetContents** Liest den Kompletten Handle aus

```
hGetContents ::  Handle -> IO String
```

**hPutChar** Schreibt ein Zeichen in einen Handle

```
hPutChar ::  Handle -> Char -> IO ()
```

**hPutStr** Schreibt einen String in einen Handle

```
hPutStr ::  Handle -> String -> IO ()
```

**hPutStrLn** Schreibt einen String und ein Newline in einen Handle

```
hPutStrLn ::  Handle -> String -> IO ()
```

**hClose** Schließt einen Handle.

```
hClose ::  Handle -> IO ()
```

# Operationen auf Handles

**hGetChar** Liest ein Zeichen von dem Handle

```
hGetChar ::  Handle -> IO Char
```

**hGetLine** Liest einen String von dem Handle (bis zum Newline)

```
hGetLine ::  Handle -> IO String
```

**hGetContents** Liest den Kompletten Handle aus

```
hGetContents ::  Handle -> IO String
```

**hPutChar** Schreibt ein Zeichen in einen Handle

```
hPutChar ::  Handle -> Char -> IO ()
```

**hPutStr** Schreibt einen String in einen Handle

```
hPutStr ::  Handle -> String -> IO ()
```

**hPutStrLn** Schreibt einen String und ein Newline in einen Handle

```
hPutStrLn ::  Handle -> String -> IO ()
```

**hClose** Schließt einen Handle.

```
hClose ::  Handle -> IO ()
```

# Operationen auf Handles

**hGetChar** Liest ein Zeichen von dem Handle

```
hGetChar ::  Handle -> IO Char
```

**hGetLine** Liest einen String von dem Handle (bis zum Newline)

```
hGetLine ::  Handle -> IO String
```

**hGetContents** Liest den Kompletten Handle aus

```
hGetContents ::  Handle -> IO String
```

**hPutChar** Schreibt ein Zeichen in einen Handle

```
hPutChar ::  Handle -> Char -> IO ()
```

**hPutStr** Schreibt einen String in einen Handle

```
hPutStr ::  Handle -> String -> IO ()
```

**hPutStrLn** Schreibt einen String und ein Newline in einen Handle

```
hPutStrLn ::  Handle -> String -> IO ()
```

**hClose** Schließt einen Handle.

```
hClose ::  Handle -> IO ()
```

# Operationen auf Handles

**hGetChar** Liest ein Zeichen von dem Handle

```
hGetChar ::  Handle -> IO Char
```

**hGetLine** Liest einen String von dem Handle (bis zum Newline)

```
hGetLine ::  Handle -> IO String
```

**hGetContents** Liest den Kompletten Handle aus

```
hGetContents ::  Handle -> IO String
```

**hPutChar** Schreibt ein Zeichen in einen Handle

```
hPutChar ::  Handle -> Char -> IO ()
```

**hPutStr** Schreibt einen String in einen Handle

```
hPutStr ::  Handle -> String -> IO ()
```

**hPutStrLn** Schreibt einen String und ein Newline in einen Handle

```
hPutStrLn ::  Handle -> String -> IO ()
```

**hClose** Schließt einen Handle.

```
hClose ::  Handle -> IO ()
```

# Datenströme sind Datenströme

Operationen klingen vertraut?

stdin, stdout und stderr sind Handles (Definiert in System.IO)

# Datenströme sind Datenströme

Operationen klingen vertraut?
stdin, stdout und stderr sind Handles (Definiert in System.IO)

# Auszug aus System.IO

Haskell: AuszugAusSystemIO.hs

```
putChar          :: Char -> IO ()
putChar c        =  hPutChar stdout c

putStr           :: String -> IO ()
putStr s         =  hPutStr stdout s

putStrLn         :: String -> IO ()
putStrLn s       =  hPutStrLn stdout s

print            :: Show a => a -> IO ()
print x          =  putStrLn (show x)

getChar          :: IO Char
getChar          =  hGetChar stdin

getLine          :: IO String
getLine          =  hGetLine stdin

getContents      :: IO String
getContents      =  hGetContents stdin
```

# Dateien öffnen

openFile Öffnet eine Datei zum Lesen/Schreiben oder Erweitern

```
openFile ::  FilePath -> IOMode -> IO Handle
```

FilePath
```
type FilePath = String
```

IOMode
```
data IOMode = ReadMode | WriteMode |
AppendMode | ReadWriteMode
```

# Dateien öffnen

openFile Öffnet eine Datei zum Lesen/Schreiben oder Erweitern

```
openFile ::  FilePath -> IOMode -> IO Handle
```

FilePath

```
type FilePath = String
```

IOMode

```
data IOMode = ReadMode | WriteMode |
AppendMode | ReadWriteMode
```

# Dateien öffnen

openFile Öffnet eine Datei zum Lesen/Schreiben oder Erweitern

```
openFile ::  FilePath -> IOMode -> IO Handle
```

FilePath

```
type FilePath = String
```

IOMode

```
data IOMode = ReadMode | WriteMode |
AppendMode | ReadWriteMode
```

# Dateien einlesen

readFile Liest eine Datei ein

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

- Liest die Datei mit `hGetContents` ein.
- Datei wird Lazy gelesen.
- Bis die Datei vollständig gelesen worden ist, ist der Handle in einem semiclosed Zustand
- Gibt Probleme, wenn man mehrfach in eine Datei schreiben und aus einer Datei Lesen möchte

# Dateien einlesen

readFile Liest eine Datei ein

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

- Liest die Datei mit `hGetContents` ein.
- Datei wird Lazy gelesen.
- Bis die Datei vollständig gelesen worden ist, ist der Handle in einem semiclosed Zustand
- Gibt Probleme, wenn man mehrfach in eine Datei schreiben und aus einer Datei Lesen möchte

# Dateien einlesen

readFile Liest eine Datei ein

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

- Liest die Datei mit `hGetContents` ein.
- Datei wird Lazy gelesen.
- Bis die Datei vollständig gelesen worden ist, ist der Handle in einem semiclosed Zustand
- Gibt Probleme, wenn man mehrfach in eine Datei schreiben und aus einer Datei Lesen möchte

# Dateien einlesen

readFile Liest eine Datei ein

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

- Liest die Datei mit `hGetContents` ein.
- Datei wird Lazy gelesen.
- Bis die Datei vollständig gelesen worden ist, ist der Handle in einem semiclosed Zustand
- Gibt Probleme, wenn man mehrfach in eine Datei schreiben und aus einer Datei Lesen möchte

# Dateien einlesen

readFile Liest eine Datei ein

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

- Liest die Datei mit `hGetContents` ein.
- Datei wird Lazy gelesen.
- Bis die Datei vollständig gelesen worden ist, ist der Handle in einem semiclosed Zustand
- Gibt Probleme, wenn man mehrfach in eine Datei schreiben und aus einer Datei Lesen möchte

# Dateien schreiben

**writeFile** Schreibt einen String in einer Datei

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

Überschreibt den gesamten Inhalt der Datei

# Dateien schreiben

**writeFile** Schreibt einen String in einer Datei

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

Überschreibt den gesamten Inhalt der Datei

# Beispiel: Verschlüsseler

Haskell: hscrypt.hs

```
import System.IO
import Data.Bits(xor)

inputfile = "inputfile"
outputfile = "outputfile"
keyfile = "keyfile"

encrypt :: String -> String -> String -> String
encryot [] _ _ = []
encrypt input [] origkey = encrypt input origkey origkey
encrypt (s:input) (k:key) origkey =
  toEnum (xor (fromEnum s) (fromEnum k)) : encrypt input key origkey

main = do
  key <- readFile keyfile
  input <- readFile inputfile
  filehandle <- openFile outputfile WriteMode
  hPutStr filehandle $ encrypt input key key
  hClose filehandle
```

# Bibliothek: Network

- Hochleveliger Zugriff auf Netzwerkfunktionen
- Bietet auch einen Lowleveligen Zugriff mit `Network.Socket`
- Netwerkströme werden als Handles aufgefasst
- Windows only Benötigt `withSocketsDo :: IO a -> IO a`

# Bibliothek: Network

- Hochleveliger Zugriff auf Netzwerkfunktionen
- Bietet auch einen Lowleveligen Zugriff mit `Network.Socket`
- Netwerkströme werden als Handles aufgefasst
- Windows only Benötigt `withSocketsDo :: IO a -> IO a`

# Bibliothek: Network

- Hochleveliger Zugriff auf Netzwerkfunktionen
- Bietet auch einen Lowleveligen Zugriff mit `Network.Socket`
- Netwerkströme werden als Handles aufgefasst
- Windows only Benötigt `withSocketsDo :: IO a -> IO a`

# Bibliothek: Network

- Hochleveliger Zugriff auf Netzwerkfunktionen
- Bietet auch einen Lowleveligen Zugriff mit `Network.Socket`
- Netwerkströme werden als Handles aufgefasst
- Windows only Benötigt `withSocketsDo :: IO a -> IO a`

# Datentypen

**PortID**

- Ein Servicename: `Service String`
  *Bsp.: Service "ftp"*
- Eine Portnummer: `PortNumber PortNumber`
  *Bsp.: PortNumber 1337*
- Unix only Ein Unix Socket: `UnixSocket String`
  *UnixSocket "/home/hammy/theSocket"*

**HostName**

- Ein Hostname *Bsp.: "haskell.org"*
- Eine IPv4 Adresse *Bsp.: "78.46.100.180"*
- Eine IPv6 Adresse *Bsp.: "2a01:4f8:121:6::10"*

# Datentypen

PortID

- Ein Servicename: `Service String`
  *Bsp.: Service "ftp"*
- Eine Portnummer: `PortNumber PortNumber`
  *Bsp.: PortNumber 1337*
- Unix only Ein Unix Socket: `UnixSocket String`
  *UnixSocket "/home/hammy/theSocket"*

HostName

- Ein Hostname *Bsp.: "haskell.org"*
- Eine IPv4 Adresse *Bsp.: "78.46.100.180"*
- Eine IPv6 Adresse *Bsp.: "2a01:4f8:121:6::10"*

# Datentypen

PortID

- Ein Servicename: `Service String`
  *Bsp.: Service "ftp"*
- Eine Portnummer: `PortNumber PortNumber`
  *Bsp.: PortNumber 1337*
- Unix only Ein Unix Socket: `UnixSocket String`
  *UnixSocket "/home/hammy/theSocket"*

HostName

- Ein Hostname *Bsp.: "haskell.org"*
- Eine IPv4 Adresse *Bsp.: "78.46.100.180"*
- Eine IPv6 Adresse *Bsp.: "2a01:4f8:121:6::10"*

# Datentypen

PortID

- Ein Servicename: `Service String`
  *Bsp.: Service "ftp"*
- Eine Portnummer: `PortNumber PortNumber`
  *Bsp.: PortNumber 1337*
- Unix only Ein Unix Socket: `UnixSocket String`
  *UnixSocket "/home/hammy/theSocket"*

HostName

- Ein Hostname *Bsp.: "haskell.org"*
- Eine IPv4 Adresse *Bsp.: "78.46.100.180"*
- Eine IPv6 Adresse *Bsp.: "2a01:4f8:121:6::10"*

# Datentypen

**PortID**

- Ein Servicename: `Service String`
  *Bsp.: Service "ftp"*
- Eine Portnummer: `PortNumber PortNumber`
  *Bsp.: PortNumber 1337*
- Unix only Ein Unix Socket: `UnixSocket String`
  *UnixSocket "/home/hammy/theSocket"*

**HostName**

- Ein Hostname *Bsp.: "haskell.org"*
- Eine IPv4 Adresse *Bsp.: "78.46.100.180"*
- Eine IPv6 Adresse *Bsp.: "2a01:4f8:121:6::10"*

# Datentypen

**PortID**

- Ein Servicename: `Service String`
  *Bsp.: Service "ftp"*
- Eine Portnummer: `PortNumber PortNumber`
  *Bsp.: PortNumber 1337*
- Unix only Ein Unix Socket: `UnixSocket String`
  *UnixSocket "/home/hammy/theSocket"*

**HostName**

- Ein Hostname *Bsp.: "haskell.org"*
- Eine IPv4 Adresse *Bsp.: "78.46.100.180"*
- Eine IPv6 Adresse *Bsp.: "2a01:4f8:121:6::10"*

# Datentypen

**PortID**

- Ein Servicename: `Service String`
  *Bsp.: Service "ftp"*
- Eine Portnummer: `PortNumber PortNumber`
  *Bsp.: PortNumber 1337*
- Unix only Ein Unix Socket: `UnixSocket String`
  *UnixSocket "/home/hammy/theSocket"*

**HostName**

- Ein Hostname *Bsp.: "haskell.org"*
- Eine IPv4 Adresse *Bsp.: "78.46.100.180"*
- Eine IPv6 Adresse *Bsp.: "2a01:4f8:121:6::10"*

# Datentypen

**PortID**

- Ein Servicename: `Service String`
  *Bsp.: Service "ftp"*
- Eine Portnummer: `PortNumber PortNumber`
  *Bsp.: PortNumber 1337*
- Unix only Ein Unix Socket: `UnixSocket String`
  *UnixSocket "/home/hammy/theSocket"*

**HostName**

- Ein Hostname *Bsp.: "haskell.org"*
- Eine IPv4 Adresse *Bsp.: "78.46.100.180"*
- Eine IPv6 Adresse *Bsp.: "2a01:4f8:121:6::10"*

# Server

**listenOn** Öffnet einen Socket auf einem Port, und lauscht

```
listenOn ::  PortID -> IO Socket
```

**accept** Wartet auf Connections auf einem Socket, und gibt einen Handle für diese Verbindung

```
accept ::  Socket -> IO (Handle, HostName,
PortNumber)
```

**sClose** Schließt einen übergebenen Socket

```
sclose ::  Socket -> IO ()
```

# Server

**listenOn** Öffnet einen Socket auf einem Port, und lauscht

```
listenOn ::  PortID -> IO Socket
```

**accept** Wartet auf Connections auf einem Socket, und gibt einen Handle für diese Verbindung

```
accept ::  Socket -> IO (Handle, HostName,
PortNumber)
```

**sClose** Schließt einen übergebenen Socket

```
sclose ::  Socket -> IO ()
```

# Server

**listenOn** Öffnet einen Socket auf einem Port, und lauscht

```
listenOn ::  PortID -> IO Socket
```

**accept** Wartet auf Connections auf einem Socket, und gibt einen Handle für diese Verbindung

```
accept ::  Socket -> IO (Handle, HostName,
PortNumber)
```

**sClose** Schließt einen übergebenen Socket

```
sclose ::  Socket -> IO ()
```

# Client

connectTo Verbindet sich zu einem Port auf einem Rechner

```
connectTo ::  HostName -> PortID -> IO
Handle
```

# Beispiel: Einfacher Datei Transfer (Client)

Haskell: hsnclient.hs

```
import Network
import System.IO
import System.Environment

main = do
    files <- getArgs
    str <- mapM readFile files
    putStrLn "eine Datei"
    handle <- connectTo "localhost" $ PortNumber 1337
    hPutStr handle $ concat str
    hClose handle
```

# Beispiel: Einfacher Datei Transfer (Server)

Haskell: hsnserver.hs

```
import Network
import System.IO

main = withSocketsDo $ do
  socket <- listenOn $ PortNumber 1337
  (handle,_,_) <- accept socket
  repeatIfNotEOF handle $ do
        line <- hGetLine handle
        putStrLn line
  where
    repeatIfNotEOF handle f = do
      isEOF <- hIsEOF handle
      if isEOF
        then return ()
        else f >> repeatIfNotEOF handle f
```

# Bibliothek: Control.Concurrent

Enthält Funktionen und Datentypen für:

- Nebenläufigkeit
- Synchronisation
- "Globale Variablen"
- Scheduling

# Bibliothek: Control.Concurrent

Enthält Funktionen und Datentypen für:

- Nebenläufigkeit
- Synchronisation
- "Globale Variablen"
- Scheduling

# Bibliothek: Control.Concurrent

Enthält Funktionen und Datentypen für:

- Nebenläufigkeit
- Synchronisation
- "Globale Variablen"
- Scheduling

# Bibliothek: Control.Concurrent

Enthält Funktionen und Datentypen für:

- Nebenläufigkeit
- Synchronisation
- "Globale Variablen"
- Scheduling

# Thread erstellen

**forkIO**

- Lässt eine übergebene Funktion in einem eigenen Thread laufen
  `forkIO :: IO () -> IO ThreadId`
- Sobald der Hauptthread beendet wird, wird das komplette Programm beendet.

Haskell: forkIO.hs

```
import Control.Concurrent

forkedThread :: Int -> IO ()
forkedThread i = (putStrLn . show) i >>= \_ -> forkedThread $ i+1

main = do
  forkIO $ forkedThread 0
  loop $ putStrLn "Ich bin der Master"
  where loop f = f >> loop f
```

# Thread erstellen

forkIO

- Lässt eine übergebene Funktion in einem eigenen Thread laufen

```
forkIO ::  IO () -> IO ThreadId
```

- Sobald der Hauptthread beendet wird, wird das komplette Programm beendet.

Haskell: forkIO.hs

```
import Control.Concurrent

forkedThread :: Int -> IO ()
forkedThread i = (putStrLn . show) i >>= \_ -> forkedThread $ i+1

main = do
  forkIO $ forkedThread 0
  loop $ putStrLn "Ich bin der Master"
  where loop f = f >> loop f
```

# Thread erstellen

forkIO

- Lässt eine übergebene Funktion in einem eigenen Thread laufen
  `forkIO :: IO () -> IO ThreadId`
- Sobald der Hauptthread beendet wird, wird das komplette Programm beendet.

Haskell: forkIO.hs

```
import Control.Concurrent

forkedThread :: Int -> IO ()
forkedThread i = (putStrLn . show) i >>= \_ -> forkedThread $ i+1

main = do
  forkIO $ forkedThread 0
  loop $ putStrLn "Ich bin der Master"
  where loop f = f >> loop f
```

# Thread erstellen

forkIO

- Lässt eine übergebene Funktion in einem eigenen Thread laufen

  `forkIO :: IO () -> IO ThreadId`
- Sobald der Hauptthread beendet wird, wird das komplette Programm beendet.

Haskell: forkIO.hs

```
import Control.Concurrent

forkedThread :: Int -> IO ()
forkedThread i = (putStrLn . show) i >>= \_ -> forkedThread $ i+1

main = do
  forkIO $ forkedThread 0
  loop $ putStrLn "Ich bin der Master"
  where loop f = f >> loop f
```

# Gemeinsame Variablen

- Kommunikation zwischen Threads
- Kommunikation innerhalb eines Threads
- "Globale Variablen"
- Locking

# Gemeinsame Variablen

- Kommunikation zwischen Threads
- Kommunikation innerhalb eines Threads
- "Globale Variablen"
- Locking

# Gemeinsame Variablen

- Kommunikation zwischen Threads
- Kommunikation innerhalb eines Threads
- "Globale Variablen"
- Locking

# Gemeinsame Variablen

- Kommunikation zwischen Threads
- Kommunikation innerhalb eines Threads
- "Globale Variablen"
- Locking

# IORef

Definiert in `Data.IORef`

**newIORef** Gibt zu einer Variable die Referenz

```
newIORef ::  a -> IO (IORef a)
```

**readIORef** Liest aus einer Referenz den Inhalt aus

```
readIORef ::  IORef a -> IO a
```

**writeIORef** Schreibt in eine Referenz

```
writeIORef ::  IORef a -> a -> IO ()
```

**modifyIORef** Wendet eine Funktion auf den Inhalt einer Referenz an

```
modifyIORef ::  IORef a -> (a -> a) -> IO
()
```

# IORef

Definiert in `Data.IORef`

**newIORef** Gibt zu einer Variable die Referenz

```
newIORef ::  a -> IO (IORef a)
```

**readIORef** Liest aus einer Referenz den Inhalt aus

```
readIORef ::  IORef a -> IO a
```

**writeIORef** Schreibt in eine Referenz

```
writeIORef ::  IORef a -> a -> IO ()
```

**modifyIORef** Wendet eine Funktion auf den Inhalt einer Referenz an

```
modifyIORef ::  IORef a -> (a -> a) -> IO
()
```

# IORef

Definiert in `Data.IORef`

**newIORef** Gibt zu einer Variable die Referenz

```
newIORef ::  a -> IO (IORef a)
```

**readIORef** Liest aus einer Referenz den Inhalt aus

```
readIORef ::  IORef a -> IO a
```

**writeIORef** Schreibt in eine Referenz

```
writeIORef ::  IORef a -> a -> IO ()
```

**modifyIORef** Wendet eine Funktion auf den Inhalt einer Referenz an

```
modifyIORef ::  IORef a -> (a -> a) -> IO ()
```

# IORef

Definiert in `Data.IORef`

**newIORef** Gibt zu einer Variable die Referenz

```
newIORef ::  a -> IO (IORef a)
```

**readIORef** Liest aus einer Referenz den Inhalt aus

```
readIORef ::  IORef a -> IO a
```

**writeIORef** Schreibt in eine Referenz

```
writeIORef ::  IORef a -> a -> IO ()
```

**modifyIORef** Wendet eine Funktion auf den Inhalt einer Referenz an

```
modifyIORef ::  IORef a -> (a -> a) -> IO
()
```

# Beispiel IORef

Haskell: IORef.hs

```
import Control.Concurrent
import Data.IORef
import Data.Char

printAllTheTime :: IORef String -> IO ()
printAllTheTime ref = do
  var <- readIORef ref
  putStrLn var
  getLine
  printAllTheTime ref

main = do
  ref <- newIORef "Foo"
  forkIO $ modifyIORef ref (map toUpper)
  printAllTheTime ref
```

# MVar

- Vorteil gegenüber IORefs: Atomares Lesen und Schreiben

**newMVar** Erstellt aus einer Variable eine MVar

```
newMVar ::  a -> IO (MVar a)
```

**takeMVar** Nimmt den Wert aus einer MVar und leert die MVar. Falls die MVar schon leer war, blockiert der Thread.

```
takeMVar ::  MVar a -> IO a
```

**putMVar** Schreibt einen Wert in eine leere MVar. Falls die MVar nicht leer ist, blockiert der Thread.

```
putMVar ::  MVar a -> a -> IO ()
```

**readMVar** Kombination aus takeMVar und putMVar

```
readMVar ::  MVar a -> IO a
```

**swapMVar** Tauscht atomar den Wert in der MVar aus.

```
swapMVar ::  MVar a -> a -> IO a
```

# MVar

- Vorteil gegenüber IORefs: Atomares Lesen und Schreiben

**newMVar** Erstellt aus einer Variable eine MVar

```
newMVar ::  a -> IO (MVar a)
```

**takeMVar** Nimmt den Wert aus einer MVar und leert die MVar. Falls die MVar schon leer war, blockiert der Thread.

```
takeMVar ::  MVar a -> IO a
```

**putMVar** Schreibt einen Wert in eine leere MVar. Falls die MVar nicht leer ist, blockiert der Thread.

```
putMVar ::  MVar a -> a -> IO ()
```

**readMVar** Kombination aus takeMVar und putMVar

```
readMVar ::  MVar a -> IO a
```

**swapMVar** Tauscht atomar den Wert in der MVar aus.

```
swapMVar ::  MVar a -> a -> IO a
```

# MVar

- Vorteil gegenüber IORefs: Atomares Lesen und Schreiben

**newMVar** Erstellt aus einer Variable eine MVar

```
newMVar ::  a -> IO (MVar a)
```

**takeMVar** Nimmt den Wert aus einer MVar und leert die MVar. Falls die MVar schon leer war, blockiert der Thread.

```
takeMVar ::  MVar a -> IO a
```

**putMVar** Schreibt einen Wert in eine leere MVar. Falls die MVar nicht leer ist, blockiert der Thread.

```
putMVar ::  MVar a -> a -> IO ()
```

**readMVar** Kombination aus takeMVar und putMVar

```
readMVar ::  MVar a -> IO a
```

**swapMVar** Tauscht atomar den Wert in der MVar aus.

```
swapMVar ::  MVar a -> a -> IO a
```

# MVar

- Vorteil gegenüber IORefs: Atomares Lesen und Schreiben

**newMVar** Erstellt aus einer Variable eine MVar

```
newMVar ::  a -> IO (MVar a)
```

**takeMVar** Nimmt den Wert aus einer MVar und leert die MVar. Falls die MVar schon leer war, blockiert der Thread.

```
takeMVar ::  MVar a -> IO a
```

**putMVar** Schreibt einen Wert in eine leere MVar. Falls die MVar nicht leer ist, blockiert der Thread.

```
putMVar ::  MVar a -> a -> IO ()
```

**readMVar** Kombination aus takeMVar und putMVar

```
readMVar ::  MVar a -> IO a
```

**swapMVar** Tauscht atomar den Wert in der MVar aus.

```
swapMVar ::  MVar a -> a -> IO a
```

# MVar

- Vorteil gegenüber IORefs: Atomares Lesen und Schreiben

**newMVar** Erstellt aus einer Variable eine MVar

```
newMVar ::  a -> IO (MVar a)
```

**takeMVar** Nimmt den Wert aus einer MVar und leert die MVar. Falls die MVar schon leer war, blockiert der Thread.

```
takeMVar ::  MVar a -> IO a
```

**putMVar** Schreibt einen Wert in eine leere MVar. Falls die MVar nicht leer ist, blockiert der Thread.

```
putMVar ::  MVar a -> a -> IO ()
```

**readMVar** Kombination aus takeMVar und putMVar

```
readMVar ::  MVar a -> IO a
```

**swapMVar** Tauscht atomar den Wert in der MVar aus.

```
swapMVar ::  MVar a -> a -> IO a
```

# MVar

- Vorteil gegenüber IORefs: Atomares Lesen und Schreiben

**newMVar** Erstellt aus einer Variable eine MVar

```
newMVar ::  a -> IO (MVar a)
```

**takeMVar** Nimmt den Wert aus einer MVar und leert die MVar. Falls die MVar schon leer war, blockiert der Thread.

```
takeMVar ::  MVar a -> IO a
```

**putMVar** Schreibt einen Wert in eine leere MVar. Falls die MVar nicht leer ist, blockiert der Thread.

```
putMVar ::  MVar a -> a -> IO ()
```

**readMVar** Kombination aus takeMVar und putMVar

```
readMVar ::  MVar a -> IO a
```

**swapMVar** Tauscht atomar den Wert in der MVar aus.

```
swapMVar ::  MVar a -> a -> IO a
```

# Beispiel: Multithreaded Chat (Client)

Haskell: hschatclient.hs

```
import Network
import Control.Concurrent
import System.IO

readAndSend h = do
  line <- getLine
  hPutStrLn h line
  hFlush h

recvAndPrint h = do
  line <- hGetLine h
  putStrLn line

main = do
  handle <- connectTo "localhost" $ PortNumber 1337
  forkIO $ loop $ readAndSend handle
  loop $ recvAndPrint handle
  where loop f = f >> loop f
```

# Beispiel: Multithreaded Chat (Server)

Haskell: hschatserver.hs

```
import Control.Concurrent
import System.IO
import Data.IORef
import Network

chatWith handle handlesref = do
  line <- hGetLine handle
  handles <- readIORef handlesref
  mapM (\h -> hPutStrLn h line >> hFlush h) handles
  chatWith handle handlesref

main = do
  socket <- listenOn $ PortNumber 1337
  handlesref <- newIORef []
  loop $ do
    (handle,_,_) <- accept socket
    handles <- readIORef handlesref
    writeIORef handlesref (handle:handles)
    forkIO $ chatWith handle handlesref
  where loop f = f >> loop f
```

# Coole Sachen

**getArgs** Holt Parameter des Programmaufrufs

```
getArgs ::  IO [String]
```

**cmdArgs** Alternative zu getArgs, parst gleichzeitig auch die Parameter

**ByteString** Schnellere Implementierung von Strings, basierend auf Arrays

**Foreign Function Interface** Ansteuerung von C Funktionen in Haskell

# Coole Sachen

**getArgs** Holt Parameter des Programmaufrufs

```
getArgs ::  IO [String]
```

**cmdArgs** Alternative zu getArgs, parst gleichzeitig auch die Parameter

**ByteString** Schnellere Implementierung von Strings, basierend auf Arrays

**Foreign Function Interface** Ansteuerung von C Funktionen in Haskell

# Coole Sachen

**getArgs** Holt Parameter des Programmaufrufs

```
getArgs ::  IO [String]
```

**cmdArgs** Alternative zu getArgs, parst gleichzeitig auch die Parameter

**ByteString** Schnellere Implementierung von Strings, basierend auf Arrays

**Foreign Function Interface** Ansteuerung von C Funktionen in Haskell

# Coole Sachen

**getArgs** Holt Parameter des Programmaufrufs
`getArgs :: IO [String]`

**cmdArgs** Alternative zu getArgs, parst gleichzeitig auch die Parameter

**ByteString** Schnellere Implementierung von Strings, basierend auf Arrays

**Foreign Function Interface** Ansteuerung von C Funktionen in Haskell