Example (interactive cp: icp.hs)

main :: IO()
main =

There is much more in the Standard IO Library
(including exception handling for IO actions)
Example (interactive cp: icp.hs)

```haskell
main :: IO()
main =
    do fromH <- readOpenFile "Copy from: " ReadMode
toH <- readOpenFile "Copy to: " WriteMode
    contents <- hGetContents fromH
    putStrLn toH contents
    hClose fromH
    hClose toH
```

Example (interactive cp: icp.hs)

```haskell
main :: IO()
main =
do fromH <- readOpenFile "Copy from: " ReadMode
toH <- readOpenFile "Copy to: " WriteMode
```

Example (interactive cp: icp.hs)

main :: IO()
main =
  do fromH <- readOpenFile "Copy from: " ReadMode
toH <- readOpenFile "Copy to: " WriteMode
    contents <- hGetContents fromH
    hPutStr toH contents
  hClose fromH
  hClose toH

readOpenFile :: String -> IOMode -> IO Handle
readOpenFile prompt mode =
  do putStrLn prompt
     name <- getLine
     handle <- openFile name mode

Executing xyz.hs

If xyz.hs contains a definition of main:
  • runhaskell xyz

Example (interactive cp: icp.hs)

main :: IO()
main =
  do fromH <- readOpenFile "Copy from: " ReadMode
toH <- readOpenFile "Copy to: " WriteMode
    contents <- hGetContents fromH
    hPutStr toH contents
  hClose fromH
  hClose toH

readOpenFile :: String -> IOMode -> IO Handle
readOpenFile prompt mode =
  do putStrLn prompt
     name <- getLine
     handle <- openFile name mode
     return handle
There is much more in the Standard IO Library (including exception handling for IO actions)

If `xyz.hs` contains a definition of `main`:
- `runhaskell xyz`

If `xyz.hs` contains a definition of `main`:
- `runhaskell xyz`
  or
- `ghc xyz` → executable file `xyz`
Word: haskell
Missed:
YOU WIN!
Input secret word: ~<CInterrupted.
Main>
Leaving GHCi.
lapnipkowid:Code nipkow$ runhaskell hangman

Word: -------
Missed:
*Clapnipkowid:Code nipkow$
lapnipkowid:Code nipkow$
lapnipkowid:Code nipkow$ runhaskell icp

icp:1:1: lexical error (UTF-8 decoding error)
lapnipkowid:Code nipkow$ runhaskell icp.hs
Copy from:

Word: -------
Missed:
*Clapnipkowid:Code nipkow$
lapnipkowid:Code nipkow$
lapnipkowid:Code nipkow$ runhaskell icp

icp:1:1: lexical error (UTF-8 decoding error)
lapnipkowid:Code nipkow$ runhaskell icp.hs
Copy from:

*Clapnipkowid:Code nipkow$
lapnipkowid:Code nipkow$ ghc icp.hs
lapnipkowid:Code nipkow$ ll icp
--wxr-xr-x 1 nipkow staff 1587280 12 Dec 07:08 icp
lapnipkowid:Code nipkow$ icp
Copy from:
Executing `xyz.hs`

If `xyz.hs` contains a definition of `main`:
- `runhaskell xyz`
- `ghc xyz -> executable file xyz`

9.2 Network I/O

Types

- **data Socket**
  
  A socket is one endpoint of a two-way communication link between two programs running on the network.
- data Socket
  A socket is one endpoint of a two-way communication link between two programs running on the network.

- data PortId = PortNumber PortNumber | ...

- data PortNumber
  
  instance Num PortNumber
  \[\Rightarrow\text{PortNumber 9000 :: PortId}\]
Server functions

- `listenOn :: PortId -> IO Socket`
  Create server side socket for specific port

- `accept :: Socket -> IO (Handle, ..., ...)`
  \(\Rightarrow\) can read/write from/to socket via handle

- `sClose :: Socket -> IO ()`
  Close socket
Initialization for Windows

withSocketsDo :: IO a -> IO a

Standard use pattern:
main = withSocketsDo $ do ...

Example (pingPong.hs)

main :: IO ()
main = withSocketsDo $ do
Example (pingPong.hs)

```haskell
main :: IO ()
main = withSocketsDo $ do
  sock <- listenOn $ PortNumber 9000
  (h, _, _) <- accept sock
  hSetBuffering h LineBuffering

loop :: Handle -> IO ()
loop h = do
  input <- hGetLine h
  if take 4 input == "quit"
    then do hPutStrLn h "goodbye!"
            hClose h
    else do hPutStrLn h ("got " ++ input)
            loop h
```

Example (pingPong.hs)

```haskell
main :: IO ()
main = withSocketsDo $ do
  sock <- listenOn $ PortNumber 9000
  (h, _, _) <- accept sock
  hSetBuffering h LineBuffering

loop :: Handle -> IO ()
loop h = do
  input <- hGetLine h
  if take 4 input == "quit"
    then do hPutStrLn h "goodbye!"
            hClose h
    else do hPutStrLn h ("got " ++ input)
            loop h
```
Example (pingPong.hs)

```haskell
main :: IO ()
main = withSocketsDo $ do
  sock <- listenOn $ PortNumber 9000
  (h, _, _) <- accept sock
  hSetBuffering h LineBuffering
  loop h
  sClose sock

loop :: Handle -> IO ()
loop h = do
  input <- hGetLine h
  if take 4 input == "quit"
    then do hPutStrLn h "goodbye!"
            hClose h
    else do hPutStrLn h ("got " ++ input)
            loop h
```

Client functions

- type HostName = String
  - For example "haskell.org" or "192.168.0.1"
Client functions

- type HostName = String
  For example "haskell.org" or "192.168.0.1"

- connectTo :: HostName -> PortId -> IO Handle
  Connect to specific port of specific host

```
Example (wGet.hs)

main :: IO()
main = withSocketsDo $ do
```

```
Example (wGet.hs)

main :: IO()
main = withSocketsDo $ do
  putStrLn "Host?"
  host <- getLine
  h <- connectTo host (PortNumber 80)
  hSetBuffering h LineBuffering
```
Example (wget.hs)

```haskell
main :: IO()
main = withSocketsDo $ do
  putStrLn "Host?"
  host <- getLine
  h <- connectTo host (PortNumber 80)
  hSetBuffering h LineBuffering
  putStrLn "Resource?"
  res <- getLine
  httpRequest h ("GET " ++ res ++ " HTTP/1.0\n")
  s <- hGetContents h
```

For more detail see

- [http://hackage.haskell.org/package/network/docs/Network-Socket.html](http://hackage.haskell.org/package/network/docs/Network-Socket.html)
Example (wGet.hs)

main :: IO()
main = withSocketsDo $ do
  putStrLn "Host?"
  host <- getLine
  h <- connectTo host (PortNumber 80)
  hSetBuffering h LineBuffering
  putStrLn "Resource?"
  res <- getLine
  putStrLn h ("GET " ++ res ++ " HTTP/1.0\n")
  s <- hGetContents h
  putStrLn s

Host?
fp.in.tum.de
Resource?
/
HTTP/1.1 302 Found
Date: Fri, 12 Dec 2014 08:29:55 GMT
Server: Apache
Location: http://www21.in.tum.de/teaching/info2/WS1415/
Content-Length: 291
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a href="http://www21.in.tum.de/teaching/info2/WS1415/">here</a>.</p>
</body></html>
Example (wGet.hs)

main :: IO()
main = withSocketsDo $ do
  putStrLn "Host?"
  host <- getLine
  h <- connectTo host (PortNumber 80)
  hSetBuffering h LineBuffering
  putStrLn "Resource?"
  res <- getLine
  hPutStrLn h ("GET " ++ res ++ " HTTP/1.0\n")
  s <- hGetContents h
  putStrLn s

For more detail see

http://hackage.haskell.org/package/network/docs/Network.html

http://hackage.haskell.org/package/network/docs/Network-Socket.html

10.1 Modules
10.1 Modules

Module = collection of type, function, class etc definitions

Purposes:
- Grouping
- Interfaces
- Division of labour
- Name space management: \texttt{M.f} vs \texttt{f}
- Information hiding

GHC: one module per file
10.1 Modules

Module = collection of type, function, class etc definitions

Purposes:
- Grouping
- Interfaces
- Division of labour
- Name space management: M.f vs f
- Information hiding

GHC: one module per file
Recommendation: module M in file M.hs

module M where  -- M must start with capital letter

module M where  -- M must start with capital letter

↑
All definitions must start in this column
module M where -- M must start with capital letter
  ↑
  All definitions must start in this column
  • Exports everything defined in M (at the top level)

Selective export:
module M (T, f, ...) where

Exporting data types

module M (T) where
data T = ...

Exporting data types
module M (T) where
data T = ...

- Exports only T, but not its constructors

module M (T(C,D,...)) where
data T = ...

- Exports T and its constructors C, D, ...

- data Socket
  A socket is one endpoint of a two-way communication link between two programs running on the network.

module M (T(C,D,...)) where
data T = ...

- Exports T and its constructors C, D, ...

module M (T(..)) where
data T = ...

- Exports T and all of its constructors
Exporting data types

module M (T) where
data T = ...

- Exports only T, but not its constructors

module M (T(C,D,...)) where
data T = ...

- Exports T and its constructors C, D, ...

module M (T(..)) where
data T = ...

- Exports T and all of its constructors

Not permitted: module M (T,C,D) where

Exporting modules

By default, modules do not export names from imported modules

module B where
import A ...

Exporting modules

By default, modules do not export names from imported modules

module B where
import A ...

⇒ B does not export f

Unless the names are mentioned in the export list
Exporting modules

By default, modules do not export names from imported modules

module B where
import A
...
⇒ B does not export f

Unless the names are mentioned in the export list

module B (f) where
import A
...

Or the whole module is exported

module B (module A) where
import A
...

Exporting modules

By default, modules do not export names from imported modules

module B where
import A
...
⇒ B does not export f

Unless the names are mentioned in the export list

module B (f) where
import A
...

Or the whole module is exported

module B (module A) where
import A
...
By default, everything that is exported is imported

module B where
import A
...

module B where
import A
...
⇒ B imports f and g

Unless an import list is specified

module B where
import A (f)
...
⇒ B imports only f

Or specific names are hidden

module B where
import A hiding (g)
...
import A
import B
import C
...
...

Where does f come from??

Clearer: qualified names

... A.f ...
import TotallyAwesomeModule

... TotallyAwesomeModule.f ...

Painful

More readable:

import qualified TotallyAwesomeModule as TAM

import TotallyAwesomeModule

... TotallyAwesomeModule.f ...

Painful

More readable:

import qualified TotallyAwesomeModule as TAM

... TAM.f ...
For the full description of the module system see the [Haskell report](#).

```
import A
import B
import C
... f ...
```

Where does \( f \) come from?!

Clearer: *qualified names*

```
... A.f ...
```

Can be enforced:

```
import qualified A

\Rightarrow\text{ must always write } A.f
```

---

**Renaming modules**

```
import TotallyAwesomeModule

... TotallyAwesomeModule.f ...
```

Painful

More readable:

```
import qualified TotallyAwesomeModule as TAM

... TAM.f ...
```

For the full description of the module system see the [Haskell report](#).
10.2 Abstract Data Types