```
putStr :: IO ()
getLine :: IO String
```

`do` blocks, which have the type of the final expression.

```
line <- getLine
```

You don't have to use `in` ... just like with list comprehensions.

Run without compilng:

```
$ runhaskell file.hs
```

Compile:

```
$ ghc --make file.hs
$ ./file
```

It wraps a pure value into an IO action to get a value of the expected type in an IO context.

... <-

```
print = putStrLn . show
```

```
when :: (Monad m) => Bool -> m () -> m ()
```

It's useful for encapsulating the *if something then do some I/O action else return ()* pattern.

```
import Control.Monad
main = do
    c <- getChar
    when (c /= ' ') $ do
        putChar c
        main
```

```
sequence :: (Monad m) => [m a] -> m [a]
```

In a `do` block at gives a more concise way or writing several consecutive `<-` extractions from an IO-returning function.

```
sequence $ map print [1..100]

mapM_ print [1..100]
```

... that result is `()`.

They are utility functions for the common task of mapping a function that returns an IO action over a list and then sequencing it.

`mapM` stores the result while `mapM_` discards it.
`forM` is like `mapM` but reverses the order of the parameters.

```
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
mapM_ :: (Monad m) => (a -> m b) -> [a] -> m ()
forM :: (Monad m) => [a] -> (a -> m b) -> m [b]
```

Control.Monad

```
forever :: (Monad m) => m a -> m b
```

It takes an IO action and repeats that action indefinitely.

Don't think of a function like putStrLn as a function that takes a string and prints it to the screen. Think of it as a function that takes a string and returns an I/O action. That I/O action will, when performed, print beautiful poetry to your terminal.

The actions are performed only when the fall into the `main` function or are the result of a `ghci` line.

In the `System.Random` module we have:

```
random :: (RandomGen g, Random a) => g -> (a, g)
```

Pure or impure code and throw exceptions, but they can only be caught in the IO part of your code.