What is the type of `putStr`?

Of `getLine`?

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

What convenience is provided for performing several IO actions in sequence?

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

Retrieve a line from stdin.

```
line <- getLine
```

What is special about `let` in `do` blocks?

This is similar to ...

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

Name two ways to run a program without loading it as a script into `ghci`.

Run without compilng:

```
$ runhaskell file.hs
```

Compile:

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

What does `return` do in Haskell?

It's sort of the opposite of ...

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

... <-

What does `ghci` use to show values on the screen?

```
print = putStrLn . show
```

Give the type signature of `when`.

What is it useful for?

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

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

Use `when` to write a function that parrots input characters until a space is hit.

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

What is the type of `sequence`?

What's it useful for?

```
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.

Use `sequence` to write an expression that prints the numbers 1 to 100.

Do the same using `mapM`/`mapM_`

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

mapM_ print [1..100]
```

`ghci` prints the result of an IO action unless ...

... that result is `()`.

What are the types of `mapM`, `mapM_`, and `forM`?

What are they useful for?

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]
```

Where are many IO-related control functions located?

Control.Monad

What is the type of `forever`?

What's it useful for?

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

It takes an IO action and repeats that action indefinitely.

What's the wrong way to think about `putStrLn`?

What's the right way?

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.

How are random numbers generated in Haskell?

In the `System.Random` module we have:

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

Where can exceptions be thrown or caught?

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