Input and Output

We've mentioned that Haskell is a purely functional language. Whereas in imperative languages you usually get things done by giving the computer a series of steps to execute, functional programming is more of defining what stuff is. In Haskell, a function can't change some state, like changing the contents of a variable (when a function changes state, we say that the function has *side-effects*). The only thing a function can do in Haskell is give us back some result based on the parameters we gave it. If a function is called two times with the same parameters, it has to return the same result. While this may seem a bit limiting when you're coming from an imperative world, we've seen that it's actually really cool. In an imperative language, you have no guarantee that a simple function that should just crunch some numbers won't burn down your house, kidnap your dog and scratch your car with a potato while crunching those numbers. For instance, when we were making a binary search tree, we didn't insert an element into a tree by modifying some tree in place. Our function for inserting into a binary search

tree actually returned a new tree, because it can't change the old one.

While functions being unable to change state is good because it helps us reason about our programs, there's one problem with that. If a function can't change anything in the world, how is it supposed to tell us what it calculated? In order to tell us what it calculated, it has to change the state of an output device (usually the state of the screen), which then emits photons that travel to our brain and change the state of our mind, man.

Do not despair, all is not lost. It turns out that Haskell actually has a really clever system for dealing with functions that have side-effects that neatly separates the part of our program that is pure and the part of our program that is impure, which does all the dirty work like talking to the keyboard and the screen. With those two parts separated, we can still reason about our pure program and take advantage of all the things that purity offers, like laziness, robustness and modularity while efficiently communicating with the outside world.

Hello, world!

Up until now, we've always loaded our functions into GHCI to test them out and play with them. We've also explored the standard library functions that way. But now, after eight or so chapters, we're finally going to write our first *real* Haskell program! Yay! And sure enough, we're going to do the good old `"hello, world"` schtick.

Hey! For the purposes of this chapter, I'm going to assume you're using a unix-y environment for learning Haskell. If you're in Windows, I'd suggest you download

Cygwin, which is a Linux-like environment for Windows, A.K.A. just what you need.

So, for starters, punch in the following in your favorite text editor:

```
main = putStrLn "hello, world"
```

We just defined a name called `main` and in it we call a function called `putStrLn` with the parameter `"hello, world"`. Looks pretty much run of the mill, but it isn't, as we'll see in just a few moments. Save that file as `helloworld.hs`.

And now, we're going to do something we've never done before. We're actually going to compile our program! I'm so excited! Open up your terminal and navigate to the directory where `helloworld.hs` is located and do the following:

```
$ ghc --make helloworld
[1 of 1] Compiling Main             ( helloworld.hs, helloworld.o )
Linking helloworld ...
```

Okay! With any luck, you got something like this and now you can run your program by doing `./helloworld`.

```
$ ./helloworld
hello, world
```

And there we go, our first compiled program that printed out something to the terminal. How extraordinarily boring!

Let's examine what we wrote. First, let's look at the type of the function `putStrLn`.

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t putStrLn "hello, world"
putStrLn "hello, world" :: IO ()
```

We can read the type of `putStrLn` like this: `putStrLn` takes a string and returns an **I/O action** that has a result type of `()` (i.e. the empty tuple, also know as unit). An I/O action is something that, when performed, will carry out an action with a side-effect (that's usually either reading from the input or printing stuff to the screen) and will also contain some kind of return value inside it. Printing a string to the terminal doesn't really have any kind of meaningful return value, so a dummy value of `()` is used.

The empty tuple is a value of `()` and it also has a type of `()`.

So, when will an I/O action be performed? Well, this is where `main` comes in. An I/O action will be performed when we give it a name of `main` and then run our program.

Having your whole program be just one I/O action seems kind of limiting. That's why we can use *do* syntax to glue together several I/O actions into one. Take a look at the following example:

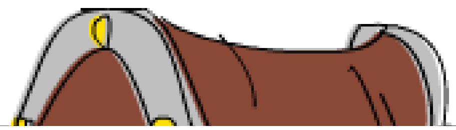
```haskell
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

Ah, interesting, new syntax! And this reads pretty much like an imperative program. If you compile it and try it out, it will probably behave just like you expect it to. Notice that we said *do* and then we laid out a series of steps, like we would in an imperative program. Each of these steps is an I/O action. By putting them together with *do* syntax, we glued them into one I/O action. The action that we got has a type of `IO ()`, because that's the type of the last I/O action inside.

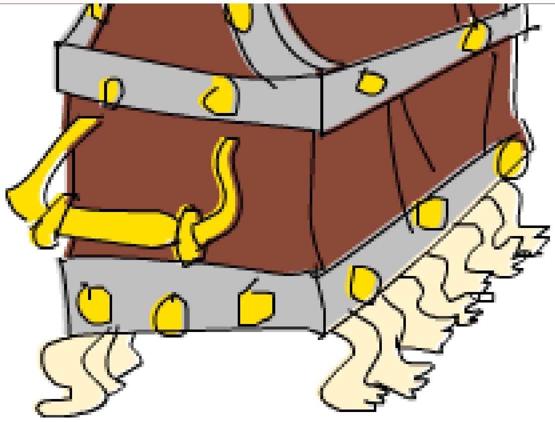
Because of that, `main` always has a type signature of `main :: IO something`, where `something` is some concrete type. By convention, we don't usually specify a type declaration for `main`.

An interesting thing that we haven't met before is the third line, which states `name <- getLine`. It looks like it reads a line from the input and stores it into a variable called `name`. Does it really? Well, let's examine the type of `getLine`.

```
ghci> :t getLine
getLine :: IO String
```



Aha, o-kay. `getLine` is an I/O action that contains a result type of `String`. That makes sense, because it will wait for the user to input something at the terminal and then that



something will be represented as a string. So what's up with `name <- getLine` then? You can read that piece of code like this: **perform the I/O action `getLine` and then bind its result value to `name`**. `getLine` has a type of `IO String`, so `name` will have a type of `String`. You can think of an I/O action as a box with little feet that will go out into the real world and do something there (like write some graffiti on a wall) and maybe bring back some data. Once it's fetched that data for you, the only way to open the box and get the data inside it is to use the `<-` construct. And if we're taking data out of an I/O action, we can only take it out when we're inside another I/O action. This is how Haskell manages to neatly separate the pure and impure parts of our code. `getLine` is in a sense impure because its result value is not guaranteed to be the same when performed twice. That's why it's sort of *tainted* with the `IO` type constructor and we can only get that data out in I/O code. And because I/O code is tainted too, any computation that depends on tainted I/O data will have a tainted result.

When I say *tainted*, I don't mean tainted in such a way that we can never use the result contained in an I/O action ever again in pure code. No, we temporarily *un-taint* the data inside an I/O action when we bind it to a name. When we do `name <- getLine`, `name` is just a normal string, because it represents what's inside the box. We can have a really complicated function that, say, takes your name (a normal string) as a parameter and tells you your fortune and your whole life's future based on your name. We can do this:

```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn $ "Read this carefully, because this is your future: " ++ tellFortune name
```

and `tellFortune` (or any of the functions it passes `name` to) doesn't have to know anything about I/O, it's just a normal `String -> String` function!

Take a look at this piece of code. Is it valid?

```
nameTag = "Hello, my name is " ++ getLine
```

If you said no, go eat a cookie. If you said yes, drink a bowl of molten lava. Just kidding, don't! The reason that this doesn't work is that `++` requires both its parameters to be lists over the same type. The left parameter has a type of `String` (or `[Char]` if you will), whilst `getLine` has a type of `IO String`. You can't concatenate a string and an I/O action. We first have to get the result out of the I/O action to get a value of type `String` and the only way to do that is to say something like `name <- getLine` inside some other I/O action. If we want to deal with impure data, we have to do it in an impure environment. So the taint of impurity spreads around much like the undead scourge and it's in our best interest to keep the I/O parts of our code as small as possible.

Every I/O action that gets performed has a result encapsulated within it. That's why our previous example program could also have been written like this:

```
main = do
    foo <- putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

However, `foo` would just have a value of `()`, so doing that would be kind of moot. Notice that we didn't bind the last `putStrLn` to anything. That's because in a *do* block, **the last action cannot be bound to a name** like the first two were. We'll see exactly why that is so a bit later when we venture off into the world of monads. For now, you can think of it in the way that the *do* block automatically extracts the value from the last action and binds it to its own result.

Except for the last line, every line in a *do* block that doesn't bind can also be written with a bind. So `putStrLn "BLAH"` can be written as `_ <- putStrLn "BLAH"`. But that's useless, so we leave out the `<-` for I/O actions that don't contain an important result, like `putStrLn something`.

Beginners sometimes think that doing

```
name = getLine
```

will read from the input and then bind the value of that to `name`. Well, it won't, all this does is give the `getLine` I/O action a different name called, well, `name`. Remember, to get the value out of an I/O action, you have to perform it inside another I/O action by binding it to a name with `<-`.

I/O actions will only be performed when they are given a name of `main` or when they're inside a bigger I/O action that we composed with a *do* block. We can also use a *do* block to glue together a few I/O actions and then we can use that I/O action in another *do* block and so on. Either way, they'll be performed only if they eventually fall into `main`.

Oh, right, there's also one more case when I/O actions will be performed. When we type out an I/O action in GHCI and press return, it will be performed.

```
ghci> putStrLn "HEEY"
HEEY
```

Even when we just punch out a number or call a function in GHCI and press return, it will evaluate it (as much as it needs) and then call `show` on it and then it will print that string to the terminal using `putStrLn` implicitly.

Remember *let* bindings? If you don't, refresh your memory on them by reading this section. They have to be in the form of `let bindings in expression`, where `bindings` are names to be given to expressions and `expression` is the expression that is to be evaluated that sees them. We also said that in list comprehensions, the *in* part isn't needed. Well, you can use them in *do* blocks pretty much like you use them in list comprehensions. Check this out:

```
import Data.Char

main = do
    putStrLn "What's your first name?"
    firstName <- getLine
    putStrLn "What's your last name?"
    lastName <- getLine
    let bigFirstName = map toUpper firstName
        bigLastName = map toUpper lastName
    putStrLn $ "hey " ++ bigFirstName ++ " " ++ bigLastName ++ ", how are you?"
```

See how the I/O actions in the *do* block are lined up? Also notice how the *let* is lined up with the I/O actions and the names of the *let* are lined up with each other? That's good practice, because indentation is important in Haskell. Now, we did `map toUpper firstName`, which turns something like `"John"` into a much cooler string like `"JOHN"`. We bound that uppercased string to a name and then used it in a string later on that we printed to the terminal.

You may be wondering when to use `<-` and when to use *let* bindings? Well, remember, `<-` is (for now) for performing I/O actions and binding their results to names. `map toUpper firstName`, however, isn't an I/O action. It's a pure expression in Haskell. So use `<-` when you want to bind results of I/O actions to names and you can use *let* bindings to bind pure expressions to names. Had we done something like `let firstName = getLine`, we would have just called the `getLine` I/O action a different name and we'd still have to run it through a `<-` to perform it.

Now we're going to make a program that continuously reads a line and prints out the same line with the words reversed. The program's execution will stop when we input a blank line. This is the program:

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ reverseWords line
            main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

To get a feel of what it does, you can run it before we go over the code.

Protip: To run a program you can either compile it and then run the produced executable file by doing `ghc --make helloworld` and then `./helloworld` or you can use the `runhaskell` command like so: `runhaskell helloworld.hs` and your program will be executed on the fly.

First, let's take a look at the `reverseWords` function. It's just a normal function that takes a string like `"hey there man"` and then calls `words` with it to produce a list of words like `["hey","there","man"]`. Then we map `reverse` on the list, getting `["yeh","ereht","nam"]` and then we put that back into one string by using `unwords` and the final result is `"yeh ereht nam"`. See how we used function composition here. Without function composition, we'd have to write something like `reverseWords st = unwords (map reverse (words st))`.

What about `main`? First, we get a line from the terminal by performing `getLine` call that line `line`. And now, we have a conditional expression. Remember that in Haskell, every *if* must have a corresponding *else* because every expression has to have some sort of value. We make the *if* so that when a condition is true (in our case, the line that we entered is blank), we perform one I/O action and when it isn't, the I/O action under the *else* is performed. That's why in an I/O *do* block, *if*s have to have a form of `if` ***`condition`*** `then` ***`I/O action`*** `else` ***`I/O action`***`.`

Let's first take a look at what happens under the *else* clause. Because, we have to have exactly one I/O action after the *else*, we use a *do* block to glue together two I/O actions into one. You could also write that part out as:

```
else (do
    putStrLn $ reverseWords line
    main)
```

This makes it more explicit that the *do* block can be viewed as one I/O action, but it's uglier. Anyway, inside the *do* block, we call `reverseWords` on the line that we got from `getLine` and then print that out to the terminal. After that, we just perform `main`. It's called recursively and that's okay, because `main` is itself an I/O action. So in a sense, we go back to the start of the program.

Now what happens when `null line` holds true? What's after the *then* is performed in that case. If we look up, we'll see that it says `then return ()`. If you've done imperative languages like C, Java or Python, you're probably thinking that you know what this `return` does and chances are you've already skipped this really long paragraph. Well, here's the thing: **the `return` in Haskell is really nothing like the `return` in most other languages!** It has the same name, which confuses a lot of people, but in reality it's quite different. In imperative languages, `return` usually ends the execution of a method or subroutine and makes it report some sort of value to whoever called it. In Haskell (in I/O actions specifically), it makes an I/O action out of a pure value. If you think about the box analogy from before, it takes a value and wraps it up in a box. The resulting I/O action doesn't actually do anything, it just has that value encapsulated as its result. So in an I/O context, `return "haha"` will have a type of `IO String`. What's the point of just transforming a pure value into an I/O action that doesn't do anything? Why taint our program with `IO` more than it has to be? Well, we needed some I/O action to carry out in the case of an empty input line. That's why we just made a bogus I/O action that doesn't do anything by writing `return ()`.

Using `return` doesn't cause the I/O *do* block to end in execution or anything like that. For instance, this program will quite happily carry out all the way to the last line:

```
main = do
    return ()
    return "HAHAHA"
    line <- getLine
    return "BLAH BLAH BLAH"
    return 4
    putStrLn line
```

All these `return`s do is that they make I/O actions that don't really do anything except have an encapsulated result and that result is thrown away because it isn't bound to a name. We can use `return` in combination with `<-` to bind stuff to names.

```
main = do
    a <- return "hell"
    b <- return "yeah!"
    putStrLn $ a ++ " " ++ b
```

So you see, `return` is sort of the opposite to `<-`. While `return` takes a value and wraps it up in a box, `<-` takes a box (and performs it) and takes the value out of it, binding it to a name. But doing this is kind of redundant, especially since you can use *let* bindings in *do* blocks to bind to names, like so:

```
main = do
```

```
    let a = "hell"
        b = "yeah"
    putStrLn $ a ++ " " ++ b
```

When dealing with I/O *do* blocks, we mostly use `return` either because we need to create an I/O action that doesn't do anything or because we don't want the I/O action that's made up from a *do* block to have the result value of its last action, but we want it to have a different result value, so we use `return` to make an I/O action that always has our desired result contained and we put it at the end.

A *do* block can also have just one I/O action. In that case, it's the same as just writing the I/O action. Some people would prefer writing `then do return ()` in this case because the *else* also has a *do*.

Before we move on to files, let's take a look at some functions that are useful when dealing with I/O.

`putStr` is much like `putStrLn` in that it takes a string as a parameter and returns an I/O action that will print that string to the terminal, only `putStr` doesn't jump into a new line after printing out the string while `putStrLn` does.

```
main = do   putStr "Hey, "
            putStr "I'm "
            putStrLn "Andy!"
```

```
$ runhaskell putstr_test.hs
```

```
Hey, I'm Andy!
```

Its type signature is `putStr :: String -> IO ()`, so the result encapsulated within the resulting I/O action is the unit. A dud value, so it doesn't make sense to bind it.

`putChar` takes a character and returns an I/O action that will print it out to the terminal.

```
main = do   putChar 't'
            putChar 'e'
            putChar 'h'
```

```
$ runhaskell putchar_test.hs
teh
```

`putStr` is actually defined recursively with the help of `putChar`. The edge condition of `putStr` is the empty string, so if we're printing an empty string, just return an I/O action that does nothing by using `return ()`. If it's not empty, then print the first character of the string by doing `putChar` and then print of them using `putStr`.

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do
    putChar x
    putStr xs
```

See how we can use recursion in I/O, just like we can use it in pure code. Just like in pure code, we define the edge case and then think what the result actually is. It's an action that first outputs the first character and then outputs the rest of the string.

`print` takes a value of any type that's an instance of `Show` (meaning that we know how to represent it as a string), calls `show` with that value to stringify it and then outputs that string to the terminal. Basically, it's just `putStrLn . show`. It first runs `show` on a value and then feeds that to `putStrLn`, which returns an I/O action that will print out our value.

```
main = do   print True
            print 2
            print "haha"
            print 3.2
            print [3,4,3]
```

```
$ runhaskell print_test.hs
True
2
"haha"
3.2
[3,4,3]
```

As you can see, it's a very handy function. Remember how we talked about how I/O actions are performed only when they fall into `main` or when we try to evaluate them in the GHCI prompt? When we type out a value (like `3` or `[1,2,3]`) and press the return key, GHCI actually uses `print` on that value to display it on our terminal!

```
ghci> 3
3
ghci> print 3
3
ghci> map (++"!") ["hey","ho","woo"]
["hey!","ho!","woo!"]
ghci> print (map (++"!") ["hey","ho","woo"])
["hey!","ho!","woo!"]
```

When we want to print out strings, we usually use `putStrLn` because we don't want the quotes around them, but for printing out values of other types to the terminal, `print` is used the most.

`getChar` is an I/O action that reads a character from the input. Thus, its type signature is `getChar :: IO Char`, because the result contained within the I/O action is a `Char`. Note that due to buffering, reading of the characters won't actually happen until the user mashes the return key.

```
main = do
    c <- getChar
    if c /= ' '
        then do
            putChar c
            main
        else return ()
```

This program looks like it should read a character and then check if it's a space. If it is, halt execution and if it isn't, print it to the terminal and then do the same thing all over again. Well, it kind of does, only not in the way you might expect. Check this out:

```
$ runhaskell getchar_test.hs
hello sir
hello
```

The second line is the input. We input `hello sir` and then press return. Due to buffering, the execution of the program will begin only when after we've hit return and not after every inputted character. But once we press return, it acts on what we've been putting in so far. Try playing with this program to get a feel for it!

The `when` function is found in `Control.Monad` (to get access to it, do `import Control.Monad`). It's interesting because in a *do* block it looks like a control flow statement, but it's actually a normal function. It takes a boolean value and an I/O action if that boolean value is `True`, it returns the same I/O action that we supplied to it. However, if it's `False`, it returns the `return ()`, action, so an I/O action that doesn't do anything. Here's how we could rewrite the previous piece of code with which we demonstrated `getChar` by using `when`:

```
import Control.Monad

main = do
    c <- getChar
    when (c /= ' ') $ do
        putChar c
        main
```

So as you can see, it's useful for encapsulating the `if something then do some I/O action else return ()` pattern.

`sequence` takes a list of I/O actions and returns an I/O actions that will perform those actions one after the other. The result contained in that I/O action will be a list of the results of all the I/O actions that were performed. Its type signature is `sequence :: [IO a] -> IO [a]`. Doing this:

```
main = do
    a <- getLine
    b <- getLine
    c <- getLine
    print [a,b,c]
```

Is exactly the same as doing this:.

```
main = do
    rs <- sequence [getLine, getLine, getLine]
    print rs
```

So `sequence [getLine, getLine, getLine]` makes an I/O action that will perform `getLine` three times. If we bind that action to a name, the result is a list of all the results, so in our case, a list of three things that the user entered at the prompt.

A common pattern with `sequence` is when we map functions like `print` or `putStrLn` over lists. Doing `map print [1,2,3,4]` won't create an I/O action. It will create a list of I/O actions, because that's like writing `[print 1, print 2, print 3, print 4]`. If we want to transform that list of I/O actions into an I/O action, we have to sequence it.

```
ghci> sequence (map print [1,2,3,4,5])
1
2
3
4
5
[(),(),(),(),()]
```

What's with the `[(),(),(),(),()]` at the end? Well, when we evaluate an I/O action in GHCI, it's performed and then its result is printed out, unless that result is `()`, in which case it's not printed out. That's why evaluating `putStrLn "hehe"` in GHCI just prints out `hehe` (because the contained result in `putStrLn "hehe"` is `()`). But when we do `getLine` in GHCI, the result of that I/O action is printed out, because `getLine` has a type of `IO String`.

Because mapping a function that returns an I/O action over a list and then sequencing it is so common, the utility functions `mapM` and `mapM_` were introduced. `mapM` takes a function and a list, maps the function over the list and then sequences it. `mapM_` does the same, only it throws away the result later. We usually use `mapM_` when we don't care what result our sequenced I/O actions have.

```
ghci> mapM print [1,2,3]
1
2
3
[(),(),()]
ghci> mapM_ print [1,2,3]
1
2
3
```

`forever` takes an I/O action and returns an I/O action that just repeats the I/O action it got forever. It's located in `Control.Monad`. This little program will indefinitely ask the user for some input and spit it back to him, CAPSLOCKED:

```
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

`forM` (located in `Control.Monad`) is like `mapM`, only that it has its parameters switched around. The first parameter is the list and the second one is the function to map over that list, which is then sequenced. Why is that useful? Well, with some creative use of lambdas and *do* notation, we can do stuff like this:

```
import Control.Monad

main = do
    colors <- forM [1,2,3,4] (\a -> do
        putStrLn $ "Which color do you associate with the number " ++ show a ++ "?"
        color <- getLine
        return color)
    putStrLn "The colors that you associate with 1, 2, 3 and 4 are: "
    mapM putStrLn colors
```

The `(\a -> do ... )` is a function that takes a number and returns an I/O action. We have to surround it with parentheses, otherwise the lambda thinks the last two I/O actions belong to it. Notice that we do `return color` in the inside *do* block. We do that so that the I/O action which the *do* block defines has the result of our color contained within it. We actually didn't have to do that, because `getLine` already has that contained within it. Doing `color <- getLine` and then `return color` is just unpacking the result from `getLine` and then repackaging it again, so it's the same as just doing `getLine`. The `forM` (called with its two parameters) produces an I/O action, whose result we bind to `colors`. `colors` is just a normal list that holds strings. At the end, we print out all those colors by doing `mapM putStrLn colors`.

You can think of `forM` as meaning: make an I/O action for every element in this list. What each I/O action will do can depend on the element that was used to make the action. Finally, perform those actions and bind their results to something. We don't have to bind it, we can also just throw it away.

```
$ runhaskell form_test.hs
Which color do you associate with the number 1?
white
```

```
Which color do you associate with the number 2?
blue
Which color do you associate with the number 3?
red
Which color do you associate with the number 4?
orange
The colors that you associate with 1, 2, 3 and 4 are:
white
blue
red
orange
```

We could have actually done that without `forM`, only with `forM` it's more readable. Normally we write `forM` when we want to map and sequence some actions that we define there on the spot using *do* notation. In the same vein, we could have replaced the last line with `forM colors putStrLn`.

In this section, we learned the basics of input and output. We also found out what I/O actions are, how they enable us to do input and output and when they are actually performed. To reiterate, I/O actions are values much like any other value in Haskell. We can pass them as parameters to functions and functions can return I/O actions as results. What's special about them is that if they fall into the `main` function (or are the result in a GHCI line), they are performed. And that's when they get to write stuff on your screen or play Yakety Sax through your speakers. Each I/O action can also encapsulate a result with which it tells you what it got from the real world.

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.

Files and streams

`getChar` is an I/O action that reads a single character from the terminal. `getLine` is an I/O action that reads a line from the terminal. These two are pretty straightforward and most programming languages have some functions or statements that are parallel to them. But now, let's meet `getContents`. `getContents` is an I/O action that reads everything from the standard input until it encounters an end-of-file character. Its type is `getContents :: IO String`. What's cool about `getContents` is that it does lazy I/O. When we do `foo <- getContents`, it doesn't read all of the input at once, store it in memory and then bind it to `foo`. No, it's lazy! It'll say: *"Yeah yeah, I'll read the input from the terminal later as we go along, when you really need it!"*.

`getContents` is really useful when we're piping the output from one program into the input of our program. In case you don't know how piping works in unix-y systems, here's a quick primer. Let's make a text file that contains the following little haiku:

```
I'm a lil' teapot
```

```
What's with that airplane food, huh?
It's so small, tasteless
```

Yeah, the haiku sucks, what of it? If anyone knows of any good haiku tutorials, let me know.

Now, recall the little program we wrote when we were introducing the `forever` function. It prompted the user for a line, returned it to him in CAPSLOCK and then did that all over again, indefinitely. Just so you don't have to scroll all the way back, here it is again:

```haskell
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

We'll save that program as `capslocker.hs` or something and compile it. And then, we're going to use a unix pipe to feed our text file directly to our little program. We're going to use the help of the GNU *cat* program, which prints out a file that's given to it as an argument. Check it out, booyaka!

```
$ ghc --make capslocker
[1 of 1] Compiling Main             ( capslocker.hs, capslocker.o )
Linking capslocker ...
```

```
$ cat haiku.txt
I'm a lil' teapot
What's with that airplane food, huh?
It's so small, tasteless
$ cat haiku.txt | ./capslocker
I'M A LIL' TEAPOT
WHAT'S WITH THAT AIRPLANE FOOD, HUH?
IT'S SO SMALL, TASTELESS
capslocker <stdin>: hGetLine: end of file
```

As you can see, piping the output of one program (in our case that was *cat*) to the input of another (*capslocker*) is done with the `|` character. What we've done is pretty much equivalent to just running *capslocker*, typing our haiku at the terminal and then issuing an end-of-file character (that's usually done by pressing Ctrl-D). It's like running *cat haiku.txt* and saying: “Wait, don't print this out to the terminal, tell it to *capslocker* instead!”.

So what we're essentially doing with that use of `forever` is taking the input and transforming it into some output. That's why we can use `getContents` to make our program even shorter and better:

```
import Data.Char

main = do
    contents <- getContents
    putStr (map toUpper contents)
```

We run the `getContents` I/O action and name the string it produces `contents`. Then, we map `toUpper` over that string and print that to the terminal. Keep in mind that because strings are basically lists, which are lazy, and `getContents` is I/O lazy, it won't try to read the whole content at once and store it into memory before printing out the capslocked version. Rather, it will print out the capslocked version as it reads it, because it will only read a line from the input when it really needs to.

```
$ cat haiku.txt | ./capslocker
I'M A LIL' TEAPOT
WHAT'S WITH THAT AIRPLANE FOOD, HUH?
IT'S SO SMALL, TASTELESS
```

Cool, it works. What if we just run *capslocker* and try to type in the lines ourselves?

```
$ ./capslocker
hey ho
HEY HO
lets go
LETS GO
```

We got out of that by pressing Ctrl-D. Pretty nice! As you can see, it prints out our capslocked input back to us line by line. When the result of `getContents` is bound to `contents`, it's not represented in memory as a real string, but more like a promise that it will produce the string eventually. When we map `toUpper` over `contents`, that's also a promise to map that function over the eventual contents. And finally when `putStr` happens, it says to the previous promise: *"Hey, I need a capslocked line!"*. It doesn't have any lines yet, so it says to `contents`: *"Hey, how about actually getting a line from the terminal?"*. So that's when `getContents` actually reads from the terminal and gives a line to the code that asked it to produce something tangible. That code then maps `toUpper` over that line and gives it to `putStr`, which prints it. And then, `putStr` says: *"Hey, I need the next line, come on!"* and this repeats until there's no more input, which is signified by an end-of-file character.

Let's make program that takes some input and prints out only those lines that are shorter than 10 characters. Observe:

```
main = do
    contents <- getContents
    putStr (shortLinesOnly contents)

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in  result
```

We've made our I/O part of the program as short as possible. Because our program is supposed to take some input and print out some output based on the input, we can implement it by reading the input contents, running a function on them and then printing out what the function gave back.

The `shortLinesOnly` function works like this: it takes a string, like `"short\nlooooooooooooooong\nshort again"`. That string has three lines, two of them are short and the middle one is long. It runs the `lines` function on that string, which converts it to `["short", "looooooooooooooong", "short again"]`, which we then bind to the name `allLines`. That list of string is then filtered so that only those lines that are shorter than 10 characters remain in the list, producing `["short", "short again"]`. And finally, `unlines` joins that list into a single newline delimited string, giving `"short\nshort again"`. Let's give it a go.

```
i'm short
so am i
i am a loooooooooong line!!!
yeah i'm long so what hahahaha!!!!!!
short line
loooooooooooooooooooooooooooong
short
```

```
$ ghc --make shortlinesonly
[1 of 1] Compiling Main             ( shortlinesonly.hs, shortlinesonly.o )
Linking shortlinesonly ...
$ cat shortlines.txt | ./shortlinesonly
i'm short
so am i
short
```

We pipe the contents of *shortlines.txt* into the output of *shortlinesonly* and as the output, we only get the short lines.

This pattern of getting some string from the input, transforming it with a function and then outputting that is so common that there exists a function which makes that even easier, called `interact`. `interact` takes a function of type `String -> String` as a parameter and returns an I/O action that will take some input, run that function on it and then print out the function's result. Let's modify our program to use that.

```
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in  result
```

Just to show that this can be achieved in much less code (even though it will be less readable) and to demonstrate our function composition skill, we're going to rework that a bit further.

```
main = interact $ unlines . filter ((<10) . length) . lines
```

Wow, we actually reduced that to just one line, which is pretty cool!

`interact` can be used to make programs that are piped some contents into them and then dump some result out or it can be used to make programs that appear to take a line of input from the user, give back some result based on that line and then take another line and so on. There isn't actually a real distinction between the two, it just depends on how the user is supposed to use them.

Let's make a program that continuously reads a line and then tells us if the line is a palindrome or not. We could just use `getLine` to read a line, tell the user if it's a palindrome and then run `main` all over again. But it's simpler if we use `interact`. When using `interact`, think about what you need to do to transform some input into the desired output. In our case, we have to replace each line of the input with either `"palindrome"` or `"not a palindrome"`. So we have to write a function that transforms something like `"elephant\nABCBA\nwhatever"` into `"not a palindrome\npalindrome\nnot a palindrome"`. Let's do this!

```
respondPalindromes contents = unlines (map (\xs -
> if isPalindrome xs then "palindrome" else "not a palindrome") (lines contents))
    where   isPalindrome xs = xs == reverse xs
```

Let's write this in point-free.

```
respondPalindromes = unlines . map (\xs -
> if isPalindrome xs then "palindrome" else "not a palindrome") . lines
    where   isPalindrome xs = xs == reverse xs
```

Pretty straightforward. First it turns something like `"elephant\nABCBA\nwhatever"` into `["elephant", "ABCBA", "whatever"]` and then it maps that lambda over it, giving `["not a palindrome", "palindrome", "not a palindrome"]` and then `unlines` joins that list into a single, newline delimited string. Now we can do

```
main = interact respondPalindromes
```

Let's test this out:

```
$ runhaskell palindromes.hs
hehe
not a palindrome
ABCBA
palindrome
cookie
not a palindrome
```

Even though we made a program that transforms one big string of input into another, it acts like we made a program that does it line by line. That's because Haskell is lazy and it wants to print the first line of the result string, but it can't because it doesn't have the first line of the input yet. So as soon as we give it the first line of input, it prints the first line of the output. We get out of the program by issuing an end-of-line character.

We can also use this program by just piping a file into it. Let's say we have this file:

```
dogaroo
radar
rotor
madam
```

and we save it as **`words.txt`**. This is what we get by piping it into our program:

```
$ cat words.txt | runhaskell palindromes.hs
not a palindrome
palindrome
palindrome
palindrome
```

Again, we get the same output as if we had run our program and put in the words ourselves at the standard input. We just don't see the input that **`palindromes.hs`** because the input came from the file and not from us typing the words in.

So now you probably see how lazy I/O works and how we can use it to our advantage. You can just think in terms of what the output is supposed to be for some given input and write a function to do that transformation. In lazy I/O, nothing is eaten from the input until it absolutely has to be because what we want to print right now depends on that input.

So far, we've worked with I/O by printing out stuff to the terminal and reading from it. But what about reading and writing files? Well, in a way, we've already been doing that. One way to think about reading from the terminal is to imagine that it's like reading from a (somewhat special) file. Same goes for writing to the terminal, it's kind of like writing to a file. We can call these two files **`stdout`** and **`stdin`**, meaning *standard output* and *standard input*, respectively. Keeping that in mind, we'll see that writing to and reading from files is very much like writing to the standard output and reading from the standard input.

We'll start off with a really simple program that opens a file called *girlfriend.txt*, which contains a verse from Avril Lavigne's #1 hit *Girlfriend*, and just prints out out to the terminal. Here's *girlfriend.txt*:

```
Hey! Hey! You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

And here's our program:

```
import System.IO

main = do
    handle <- openFile "girlfriend.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

Running it, we get the expected result:

```
$ runhaskell girlfriend.hs
Hey! Hey! You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

Let's go over this line by line. The first line is just four exclamations, to get our attention. In the second line, Avril tells us that she doesn't like our current romantic partner. The third line serves to emphasize that disapproval, whereas the fourth line suggests we should seek out a new girlfriend.

Let's also go over the program line by line! Our program is several I/O actions glued together with a *do* block. In the first line of the *do* block, we notice a new function called `openFile`. This is its type signature: `openFile :: FilePath -> IOMode -> IO Handle`. If you read that out loud, it states: `openFile` takes a file path and an `IOMode` and returns an I/O action that will open a file and have the file's associated handle encapsulated as its result.

`FilePath` is just a type synonym for `String`, simply defined as:

```
type FilePath = String
```

`IOMode` is a type that's defined like this:

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

Just like our type that represents the seven possible values for the days of the week, this type is an enumeration that represents what we want to do with our opened file. Very simple. Just note that this type is `IOMode` and not `IO Mode`. `IO Mode` would be the type of an I/O action that has a value of some type `Mode` as its result, but `IOMode` is just a simple enumeration.

Finally, it returns an I/O action that will open the specified file in the specified mode. If we bind that action to something we get a `Handle`. A value of type `Handle` represents where our file is. We'll use that handle so we know which file to read from. It would be stupid to read a file but not bind that read to a handle because we wouldn't be able to do anything with the file. So in our case, we bound the handle to `handle`.

In the next line, we see a function called `hGetContents`. It takes a `Handle`, so it knows which file to get the contents from and returns an `IO String` — an I/O action that holds as its result the contents of the file. This function is pretty much like `getContents`. The only difference is that `getContents` will automatically read from the standard input (that is from the terminal), whereas `hGetContents` takes a file handle which tells it which file to read from. In all other respects, they work the same. And just like `getContents`, `hGetContents` won't attempt to read the file at once and store it in memory, but it will read it as needed. That's really cool because we can treat `contents` as the whole contents of the file, but it's not really loaded in memory. So if this were a really huge file, doing `hGetContents` wouldn't choke up our memory, but it would read only what it needed to from the file, when it needed to.

Note the difference between the handle used to identify a file and the contents of the file, bound in our program to `handle` and `contents`. The handle is just something by which we know what our file is. If you imagine your whole file system to be a really big book and each file is a chapter in the book, the handle is a bookmark that shows where you're currently reading (or writing) a chapter, whereas the contents are the actual chapter.

With `putStr contents` we just print the contents out to the standard output and then we do `hClose`, which takes a handle and returns an I/O action that closes the file. You have to close the file yourself after opening it with `openFile`!

Another way of doing what we just did is to use the `withFile` function, which has a type signature of `withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a`. It takes a path to a file, an `IOMode` and then it takes a function that takes a handle and returns some I/O action. What it returns is an I/O action that will open that file, do something we want with the file and then close it. The result encapsulated in the final I/O action that's returned is the same as the result of the I/O action that the function we give it returns. This might sound a bit complicated, but it's really simple, especially with lambdas, here's our previous example rewritten to use `withFile`:

```haskell
import System.IO

main = do
    withFile "girlfriend.txt" ReadMode (\handle -> do
        contents <- hGetContents handle
        putStr contents)
```

As you can see, it's very similar to the previous piece of code. `(\handle -> ... )` is the function that takes a handle and returns an I/O action and it's usually done like this, with a lambda. The reason it has to take a function that returns an I/O action instead of just taking an I/O action to do and then close the file is because the I/O action that we'd pass to it wouldn't know on which file to operate. This way, `withFile` opens the file and then passes the handle to the function we gave it. It gets an I/O action back from that function and then makes an I/O action that's just like it, only it closes the file afterwards. Here's how we can make our own `withFile` function:

```haskell
withFile' :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile' path mode f = do
    handle <- openFile path mode
    result <- f handle
    hClose handle
    return result
```

We know the result will be an I/O action so we can just start off with a *do*. First we open the file and get a handle from it. Then, we apply `handle` to our function to get back the I/O action that does all the work. We bind that action to `result`, close the handle and then do `return result`. By `return`ing the result encapsulated in the I/O action that we got from `f`, we make it so that our I/O action encapsulates the same result as the one we got from `f handle`. So if `f handle` returns an action that will read a number of lines from the standard input and write them to a file and have as its result encapsulated the number of lines it read, if we used that with `withFile'`, the resulting I/O action would also have as its result the number of lines read.

Just like we have `hGetContents` that works like `getContents` but for a specific file, there's also `hGetLine`, `hPutStr`, `hPutStrLn`, `hGetChar`, etc. They work just like their counterparts without the *h*, only they take a handle as a parameter and operate on that specific file instead of operating on standard input or standard output. Example: `putStrLn` is a function that takes a string and returns an I/O action that will print out that string to the terminal and a newline after it. `hPutStrLn` takes a handle and a string and returns an I/O action that will write that string to the file associated with the handle and then put a newline after it. In the same vein, `hGetLine` takes a handle and returns an I/O action that reads a line from its file.

Loading files and then treating their contents as strings is so common that we have these three nice little functions to make our work even easier:

`readFile` has a type signature of `readFile :: FilePath -> IO String`. Remember, `FilePath` is just a fancy name for `String`. `readFile` takes a path to a file and returns an I/O action that will read that file (lazily, of course) and bind its contents to something as a string. It's usually more handy than doing `openFile` and binding it to a handle and then doing `hGetContents`. Here's how we could have written our previous example with `readFile`:

```
import System.IO

main = do
    contents <- readFile "girlfriend.txt"
    putStr contents
```

Because we don't get a handle with which to identify our file, we can't close it manually, so Haskell does that for us when we use `readFile`.

`writeFile` has a type of `writeFile :: FilePath -> String -> IO ()`. It takes a path to a file and a string to write to that file and returns an I/O action that will do the writing. If such a file already exists, it will be stomped down to zero length before being written on. Here's how to turn *girlfriend.txt* into a CAPSLOCKED version and write it to *girlfriendcaps.txt*:

```
import System.IO
import Data.Char

main = do
    contents <- readFile "girlfriend.txt"
    writeFile "girlfriendcaps.txt" (map toUpper contents)
```

```
$ runhaskell girlfriendtocaps.hs
$ cat girlfriendcaps.txt
HEY! HEY! YOU! YOU!
I DON'T LIKE YOUR GIRLFRIEND!
NO WAY! NO WAY!
I THINK YOU NEED A NEW ONE!
```

`appendFile` has a type signature that's just like **`writeFile`**, only **`appendFile`** doesn't truncate the file to zero length if it already exists but it appends stuff to it.

Let's say we have a file *todo.txt* that has one task per line that we have to do. Now let's make a program that takes a line from the standard input and adds that to our to-do list.

```
import System.IO
```

```
main = do
    todoItem <- getLine
    appendFile "todo.txt" (todoItem ++ "\n")
```

```
$ runhaskell appendtodo.hs
Iron the dishes
$ runhaskell appendtodo.hs
Dust the dog
$ runhaskell appendtodo.hs
Take salad out of the oven
$ cat todo.txt
Iron the dishes
Dust the dog
Take salad out of the oven
```

We needed to add the `"\n"` to the end of each line because `getLine` doesn't give us a newline character at the end.

Ooh, one more thing. We talked about how doing `contents <- hGetContents handle` doesn't cause the whole file to be read at once and stored in-memory. It's I/O lazy, so doing this:

```
main = do
    withFile "something.txt" ReadMode (\handle -> do
        contents <- hGetContents handle
        putStr contents)
```

is actually like connecting a pipe from the file to the output. Just like you can think of lists as streams, you can also think of files as streams. This will read one line at a time and print it out to the terminal as it goes along. So you may be asking, how wide is this pipe then? How often will the disk be accessed? Well, for text files, the default buffering is line-buffering usually. That means that the smallest part of the file to be read at once is one line. That's why in this case it actually reads a line, prints it to the output, reads the next line, prints it, etc. For binary files, the default buffering is usually block-buffering. That means that it will read the file chunk by chunk. The chunk size is some size that your operating system thinks is cool.

You can control how exactly buffering is done by using the `hSetBuffering` function. It takes a handle and a `BufferMode` and returns an I/O action that sets the buffering. `BufferMode` is a simple enumeration data type and the possible values it can hold are: `NoBuffering`, `LineBuffering` or `BlockBuffering (Maybe Int)`. The `Maybe Int` is for how big the chunk should be, in bytes. If it's `Nothing`, then the operating system determines the chunk size. `NoBuffering` means that it will be read one character at a time. `NoBuffering` usually sucks as a buffering mode because it has to access the disk so much.

Here's our previous piece of code, only it doesn't read it line by line but reads the whole file in chunks of 2048 bytes.

```haskell
main = do
    withFile "something.txt" ReadMode (\handle -> do
        hSetBuffering handle $ BlockBuffering (Just 2048)
        contents <- hGetContents handle
        putStr contents)
```

Reading files in bigger chunks can help if we want to minimize disk access or when our file is actually a slow network resource.

We can also use `hFlush`, which is a function that takes a handle and returns an I/O action that will flush the buffer of the file associated with the handle. When we're doing line-buffering, the buffer is flushed after every line. When we're doing block-buffering, it's after we've read a chunk. It's also flushed after closing a handle. That means that when we've reached a newline character, the reading (or writing) mechanism reports all the data so far. But we can use **`hFlush`** to force that reporting of data that has been read so far. After flushing, the data is available to other programs that are running at the same time.

Think of reading a block-buffered file like this: your toilet bowl is set to flush itself after it has one gallon of water inside it. So you start pouring in water and once the gallon mark is reached, that water is automatically flushed and the data in the water that you've poured in so far is read. But you can flush the toilet manually too by pressing the button on the toilet. This makes the toilet flush and all the water (data) inside the toilet is read. In case you haven't noticed, flushing the toilet manually is a metaphor for **`hFlush`**. This is not a very great analogy by programming analogy standards, but I wanted a real world object that can be flushed for the punchline.

We already made a program to add a new item to our to-do list in *todo.txt*, now let's make a program to remove an item. I'll just paste the code and then we'll go over the program together so you see that it's really easy. We'll be using a few new functions from **`System.Directory`** and one new function from **`System.IO`**, but they'll all be explained.

Anyway, here's the program for removing an item from *todo.txt*:

```haskell
import System.IO
import System.Directory
import Data.List
```

```haskell
main = do
    handle <- openFile "todo.txt" ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStrLn "These are your TO-DO items:"
    putStr $ unlines numberedTasks
    putStrLn "Which one do you want to delete?"
    numberString <- getLine
    let number = read numberString
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile "todo.txt"
    renameFile tempName "todo.txt"
```

At first, we just open *todo.txt* in read mode and bind its handle to `handle`.

Next up, we use a function that we haven't met before which is from `System.IO` — `openTempFile`. Its name is pretty self-explanatory. It takes a path to a temporary directory and a template name for a file and opens a temporary file. We used `"."` for the temporary directory, because `.` denotes the current directory on just about any OS. We used `"temp"` as the template name for the temporary file, which means that the temporary file will be named *temp* plus some random characters. It returns an I/O action that makes the temporary file and the result in that I/O action is a pair of values: the name of the temporary file and a handle. We could just open a normal file called *todo2.txt* or something like that but it's better practice to use `openTempFile` so you know you're probably not overwriting anything.

The reason we didn't use `getCurrentDirectory` to get the current directory and then pass it to `openTempFile` but instead just passed `"."` to `openTempFile` is because `.` refers to the current directory on unix-like system and Windows

Next up, we bind the contents of *todo.txt* to `contents`. Then, split that string into a list of strings, each string one line. So `todoTasks` is now something like `["Iron the dishes", "Dust the dog", "Take salad out of the oven"]`. We zip the numbers from 0 onwards and that list with a function that takes a number, like 3, and a string, like `"hey"` and returns `"3 - hey"`, so `numberedTasks` is `["0 - Iron the dishes", "1 - Dust the dog" ...`. We join that list of strings into a single newline delimited string with `unlines` and print that string out to the terminal. Note that instead of doing that, we could have also done `mapM putStrLn numberedTasks`

We ask the user which one they want to delete and wait for them to enter a number. Let's say they want to delete number 1, which is `Dust the dog`, so they punch in `1`. `numberString` is now `"1"` and because we want a number, not a string, we run `read` on that to get `1` and bind that to `number`.

Remember the `delete` and `!!` functions from `Data.List`. `!!` returns an element from a list with some index and `delete` deletes the first occurence of an element in a list and returns a new list without that occurence. `(todoTasks !! number)` (number is now `1`) returns `"Dust the dog"`. We bind `todoTasks` without the first occurence of `"Dust the dog"` to `newTodoItems` and then join that into a single string with `unlines` before writing it to the temporary file that we opened. The old file is now unchanged and the temporary file contains all the lines that the old one does, except the one we deleted.

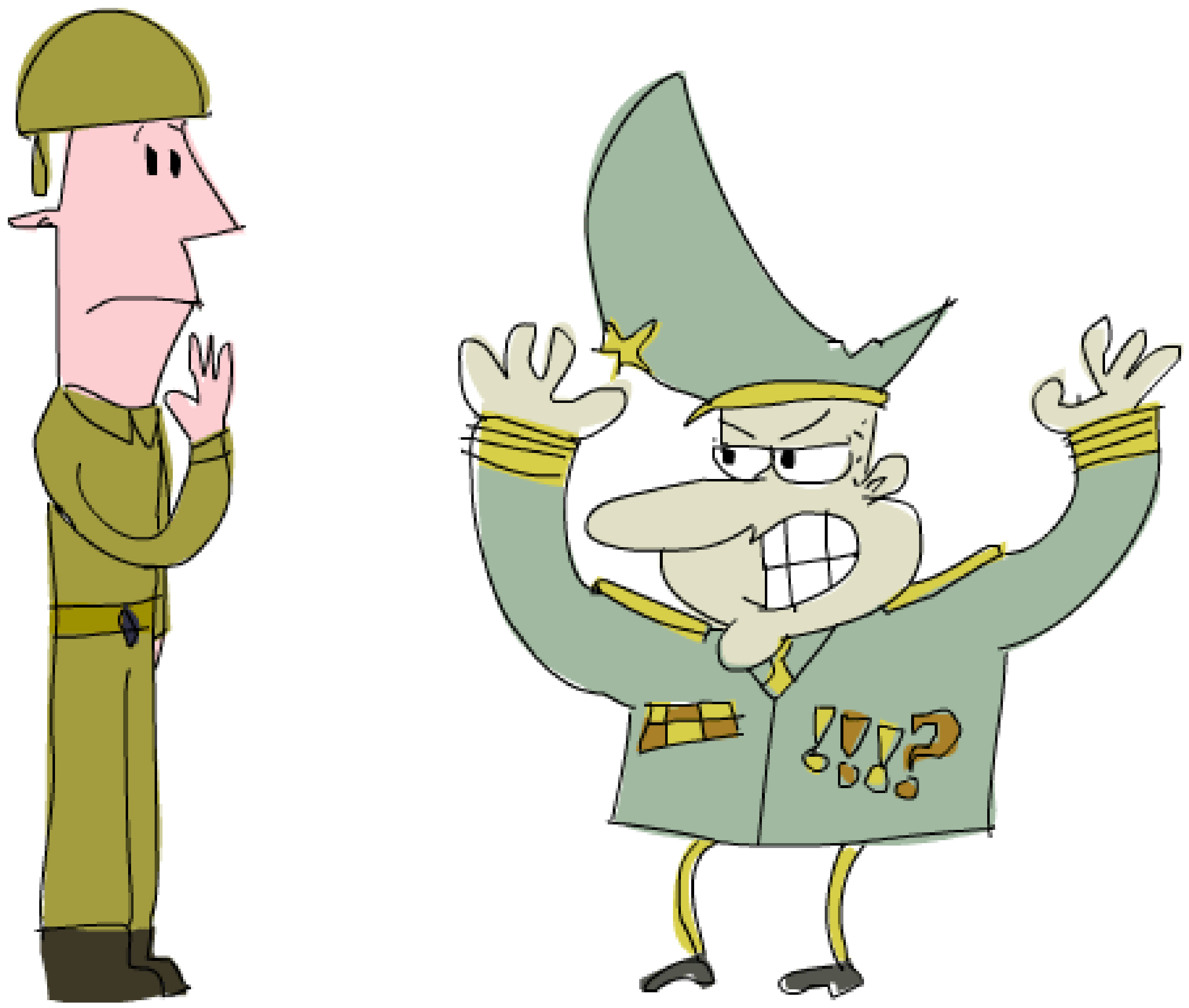
After that we close both the original and the temporary files and then we remove the original one with `removeFile`, which, as you can see, takes a path to a file and deletes it. After deleting the old *todo.txt*, we use `renameFile` to rename the temporary file to *todo.txt*. Be careful, `removeFile` and `renameFile` (which are both in `System.Directory` by the way) take file paths as their parameters, not handles.

And that's that! We could have done this in even fewer lines, but we were very careful not to overwrite any existing files and politely asked the operating system to tell us where we can put our temporary file. Let's give this a go!

```
$ runhaskell deletetodo.hs
These are your TO-DO items:
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
Which one do you want to delete?
1

$ cat todo.txt
Iron the dishes
Take salad out of the oven

$ runhaskell deletetodo.hs
These are your TO-DO items:
0 - Iron the dishes
1 - Take salad out of the oven
Which one do you want to delete?
0

$ cat todo.txt
Take salad out of the oven
```

Command line arguments

Dealing with command line arguments is pretty much a necessity if you want to make a script or application that runs on a terminal. Luckily, Haskell's standard library has a nice way of getting command line arguments of a program.



In the previous section, we made one program for adding a to-do item to our to-do list and one program for removing an item. There are two problems with the approach we took. The first one is that we just hardcoded the name of our to-do file in our code. We just decided that the file will be named *todo.txt* and that the user will never have a need for managing several to-do lists.

One way to solve that is to always ask the user which file they want to use as their to-do list. We used that approach when we wanted to know which item the user wants to delete. It works, but it's not so good, because it requires the user to run the program, wait for the program to ask something and then tell that to the program. That's called an interactive program and the difficult bit with interactive command line programs is this — what if you want to automate the execution of that program, like with a batch script? It's harder to make a batch script that interacts with a program than a batch script that just calls one program or several of them.

That's why it's sometimes better to have the user tell the program what they want when they run the program, instead of having the program ask the user once it's run. And what better way to have the user tell the program what they want it to do when they run it than via command line arguments!

The `System.Environment` module has two cool I/O actions. One is `getArgs`, which has a type of `getArgs :: IO [String]` and is an I/O action that will get the arguments that the program was run with and have as its contained result a list with the arguments. `getProgName` has a type of `getProgName :: IO String` and is an I/O action that contains the program name.

Here's a small program that demonstrates how these two work:

```
import System.Environment
import Data.List

main = do
   args <- getArgs
   progName <- getProgName
   putStrLn "The arguments are:"
   mapM putStrLn args
   putStrLn "The program name is:"
   putStrLn progName
```

We bind `getArgs` and `progName` to `args` and `progName`. We say `The arguments are:` and then for every argument in `args`, we do `putStrLn`. Finally, we also print out the program name. Let's compile this as `arg-test`.

```
$ ./arg-test first second w00t "multi word arg"
The arguments are:
first
second
w00t
multi word arg
The program name is:
arg-test
```

Nice. Armed with this knowledge you could create some cool command line apps. In fact, let's go ahead and make one. In the previous section, we made a separate program for adding tasks and a separate program for deleting them. Now, we're going to join that into one program, what it does will depend on the command line arguments. We're also going to make it so it can operate on different files, not just *todo.txt*.

We'll call it simply *todo* and it'll be able to do (haha!) three different things:

- View tasks
- Add tasks
- Delete tasks

We're not going to concern ourselves with possible bad input too much right now.

Our program will be made so that if we want to add the task `Find the magic sword of power` to the file *todo.txt*, we have to punch in `todo add todo.txt "Find the magic sword of power"` in our terminal. To view the tasks we'll just do `todo view todo.txt` and to remove the task with the index of 2, we'll do `todo remove todo.txt 2`.

We'll start by making a dispatch association list. It's going to be a simple association list that has command line arguments as keys and functions as their corresponding values. All these functions will be of type `[String] -> IO ()`. They're going to take the argument list as a parameter and return an I/O action that does the viewing, adding, deleting, etc.

```
import System.Environment
import System.Directory
import System.IO
import Data.List

dispatch :: [(String, [String] -> IO ())]
dispatch =  [ ("add", add)
            , ("view", view)
            , ("remove", remove)
            ]
```

We have yet to define `main`, `add`, `view` and `remove`, so let's start with `main`:

```
main = do
    (command:args) <- getArgs
    let (Just action) = lookup command dispatch
    action args
```

First, we get the arguments and bind them to `(command:args)`. If you remember your pattern matching, this means that the first argument will get bound to `command` and the rest of them will get bound to `args`. If we call our program like `todo add todo.txt "Spank the monkey"`, `command` will be `"add"` and `args` will be `["todo.xt", "Spank the monkey"]`.

In the next line, we look up our command in the dispatch list. Because `"add"` points to `add`, we get `Just add` as a result. We use pattern matching again to extract our function out of the `Maybe`. What happens if our command isn't in the dispatch list? Well then the lookup will return `Nothing`, but we said we won't concern ourselves with failing gracefully too much, so the pattern matching will fail and our program will throw a fit.

Finally, we call our `action` function with the rest of the argument list. That will return an I/O action that either adds an item, displays a list of items or deletes an item and because that action is part of the `main` *do* block, it will get performed. If we follow our concrete example so far and our `action` function is `add`, it will get called with `args` (so `["todo.txt", "Spank the monkey"]`) and return an I/O action that adds `Spank the monkey` to *todo.txt*.

Great! All that's left now is to implement `add`, `view` and `remove`. Let's start with `add`:

```
add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")
```

If we call our program like `todo add todo.txt "Spank the monkey"`, the `"add"` will get bound to `command` in the first pattern match in the `main` block, whereas `["todo.txt", "Spank the monkey"]` will get passed to the function that we get from the dispatch list. So, because we're not dealing with bad input right now, we just pattern match against a list with those two elements right away and return an I/O action that appends that line to the end of the file, along with a newline character.

Next, let's implement the list viewing functionality. If we want to view the items in a file, we do `todo view todo.txt`. So in the first pattern match, `command` will be `"view"` and `args` will be `["todo.txt"]`.

```
view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStr $ unlines numberedTasks
```

We already did pretty much the same thing in the program that only deleted tasks when we were displaying the tasks so that the user can choose one for deletion, only here we just display the tasks.

And finally, we're going to implement `remove`. It's going to be very similar to the program that only deleted the tasks, so if you don't understand how deleting an item here works, check out the explanation under that program. The main difference is that we're not hardcoding *todo.txt* but getting it as an argument. We're also not prompting the user for the task number to delete, we're getting it as an argument.

```
remove :: [String] -> IO ()
remove [fileName, numberString] = do
    handle <- openFile fileName ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let number = read numberString
        todoTasks = lines contents
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile fileName
    renameFile tempName fileName
```

We opened up the file based on `fileName` and opened a temporary file, deleted the line with the index that the user wants to delete, wrote that to the temporary file, removed the original file and renamed the temporary file back to `fileName`.

Here's the whole program at once, in all its glory!

```
import System.Environment
import System.Directory
import System.IO
import Data.List

dispatch :: [(String, [String] -> IO ())]
dispatch =  [ ("add", add)
            , ("view", view)
```

```haskell
            , ("remove", remove)
            ]

main = do
    (command:args) <- getArgs
    let (Just action) = lookup command dispatch
    action args

add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")

view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStr $ unlines numberedTasks

remove :: [String] -> IO ()
remove [fileName, numberString] = do
    handle <- openFile fileName ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let number = read numberString
        todoTasks = lines contents
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile fileName
    renameFile tempName fileName
```

To summarize our solution: we made a dispatch association that maps from commands to functions that take some command line arguments and return an I/O action. We see what the command is and based on that we get the appropriate function from the dispatch list. We call that function with the rest of the command line arguments to get back an I/O action that will do the appropriate thing and then just perform that action!

In other languages, we might have implemented this with a big switch case statement or whatever, but using higher order functions allows us to just tell the dispatch list to give us the appropriate function and then tell that function to give us an I/O action for some command line arguments.

Let's try our app out!

```
$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven

$ ./todo add todo.txt "Pick up children from drycleaners"

$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
3 - Pick up children from drycleaners
```

```
$ ./todo remove todo.txt 2

$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Pick up children from drycleaners
```

Another cool thing about this is that it's easy to add extra functionality. Just add an entry in the dispatch association list and implement the corresponding function and you're laughing! As an exercise, you can try implementing a `bump` function that will take a file and a task number and return an I/O action that bumps that task to the top of the to-do list.

You could make this program fail a bit more gracefully in case of bad input (for example, if someone runs `todo UP YOURS HAHAHAHA`) by making an I/O action that just reports there has been an error (say, `errorExit :: IO ()`) and then check for possible erronous input and if there is erronous input, perform the error reporting I/O action. Another way is to use exceptions, which we will meet soon.

Randomness

Many times while programming, you need to get some random data. Maybe you're making a game where a die needs to be thrown or you need to generate some test data to test out your program. There are a lot of uses for random data when programming. Well, actually, pseudo-random, because we all know that the only true source of randomness is a monkey on a unicycle

with a cheese in one hand and its butt in the other. In this section, we'll take a look at how to make Haskell generate seemingly random data.

In most other programming languages, you have functions that give you back some random number. Each time you call that function, you get back a (hopefully) different random number. How about Haskell? Well, remember, Haskell is a pure functional language. What that means is that it has referential transparency. What THAT means is that a function, if given the same parameters twice, must produce the same result twice. That's really cool because it allows us to reason differently about programs and it enables us to defer evaluation until we really need it. If I call a function, I can be sure that it won't do any funny stuff before giving me the results. All that matters are its results. However, this makes it a bit tricky for getting random numbers. If I have a function like this:

```haskell
randomNumber :: (Num a) => a
randomNumber = 4
```

It's not very useful as a random number function because it will always return `4`, even though I can assure you that the 4 is completely random, because I used a die to determine it.

How do other languages make seemingly random numbers? Well, they take various info from your computer, like the current time, how much and where you moved your mouse and what kind of noises you made behind your computer and based on that, give a number that looks really random. The combination of those factors (that randomness) is probably different in any given moment in time, so you get a different random number.

Ah. So in Haskell, we can make a random number then if we make a function that takes as its parameter that randomness and based on that returns some number (or other data type).

Enter the `System.Random` module. It has all the functions that satisfy our need for randomness. Let's just dive into one of the functions it exports then, namely random. Here's its type: `random :: (RandomGen g, Random a) => g -> (a, g)`. Whoa! Some new typeclasses in this type declaration up in here! The RandomGen typeclass is for types that can act as sources of randomness. The Random typeclass is for things that can take on random values. A boolean value can take on a random value, namely `True` or `False`. A number can also take up a plethora of different random values. Can a function take on a random value? I don't think so, probably not! If we try to translate the type declaration of `random` to English, we get something like: it takes a random generator (that's our source of randomness) and returns a random value and a new random generator. Why does it also return a new generator as well as a random value? Well, we'll see in a moment.

To use our `random` function, we have to get our hands on one of those random generators. The `System.Random` module exports a cool type, namely StdGen that is an instance of the `RandomGen` typeclass. We can either make a `StdGen` manually or we can tell the system to give us one based on a multitude of sort of random stuff.

To manually make a random generator, use the mkStdGen function. It has a type of `mkStdGen :: Int -> StdGen`. It takes an integer and based on that, gives us a random generator. Okay then, let's try using `random` and `mkStdGen` in tandem to get a (hardly random) number.

```
ghci> random (mkStdGen 100)
```

```
<interactive>:1:0:
    Ambiguous type variable `a' in the constraint:
      `Random a' arising from a use of `random' at <interactive>:1:0-20
    Probable fix: add a type signature that fixes these type variable(s)
```

What's this? Ah, right, the `random` function can return a value of any type that's part of the `Random` typeclass, so we have to inform Haskell what kind of type we want. Also let's not forget that it returns a random value and a random generator in a pair.

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

Finally! A number that looks kind of random! The first component of the tuple is our number whereas the second component is a textual representation of our new random generator. What happens if we call `random` with the same random generator again?

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

Of course. The same result for the same parameters. So let's try giving it a different random generator as a parameter.

```
ghci> random (mkStdGen 949494) :: (Int, StdGen)
(539963926,466647808 1655838864)
```

Alright, cool, great, a different number. We can use the type annotation to get different types back from that function.

```
ghci> random (mkStdGen 949488) :: (Float, StdGen)
(0.8938442,1597344447 1655838864)
ghci> random (mkStdGen 949488) :: (Bool, StdGen)
(False,1485632275 40692)
ghci> random (mkStdGen 949488) :: (Integer, StdGen)
(1691547873,1597344447 1655838864)
```

Let's make a function that simulates tossing a coin three times. If `random` didn't return a new generator along with a random value, we'd have to make this function take three random generators as a parameter and then return coin tosses for each of them. But that sounds wrong because if one generator can make a random value of type `Int` (which can take on a load of different values), it should be able to make three coin tosses (which can take on precisely eight combinations). So this is where `random` returning a new generator along with a value really comes in handy.

We'll represent a coin with a simple `Bool`. `True` is tails, `False` is heads.

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
    let (firstCoin, newGen) = random gen
        (secondCoin, newGen') = random newGen
        (thirdCoin, newGen'') = random newGen'
    in  (firstCoin, secondCoin, thirdCoin)
```

We call `random` with the generator we got as a parameter to get a coin and a new generator. Then we call it again, only this time with our new generator, to get the second coin. We do the same for the third coin. Had we called it with the same generator every time, all the coins would have had the same value and we'd only be able to get `(False, False, False)` or `(True, True, True)` as a result.

```
ghci> threeCoins (mkStdGen 21)
(True,True,True)
ghci> threeCoins (mkStdGen 22)
(True,False,True)
ghci> threeCoins (mkStdGen 943)
(True,False,True)
ghci> threeCoins (mkStdGen 944)
(True,True,True)
```

Notice that we didn't have to do `random gen :: (Bool, StdGen)`. That's because we already specified that we want booleans in the type declaration of the function. That's why Haskell can infer that we want a boolean value in this case.

So what if we want to flip four coins? Or five? Well, there's a function called `randoms` that takes a generator and returns an infinite sequence of values based on that generator.

```
ghci> take 5 $ randoms (mkStdGen 11) :: [Int]
[-1807975507,545074951,-1015194702,-1622477312,-502893664]
ghci> take 5 $ randoms (mkStdGen 11) :: [Bool]
[True,True,True,True,False]
ghci> take 5 $ randoms (mkStdGen 11) :: [Float]
[7.904789e-2,0.62691015,0.26363158,0.12223756,0.38291094]
```

Why doesn't `randoms` return a new generator as well as a list? We could implement the `randoms` function very easily like this:

```
randoms' :: (RandomGen g, Random a) => g -> [a]
randoms' gen = let (value, newGen) = random gen in value:randoms' newGen
```

A recursive definition. We get a random value and a new generator from the current generator and then make a list that has the value as its head and random numbers based on the new generator as its tail. Because we have to be able to potentially generate an infinite amount of numbers, we can't give the new random generator back.

We could make a function that generates a finite stream of numbers and a new generator like this:

```
finiteRandoms :: (RandomGen g, Random a, Num n) => n -> g -> ([a], g)
finiteRandoms 0 gen = ([], gen)
finiteRandoms n gen =
    let (value, newGen) = random gen
        (restOfList, finalGen) = finiteRandoms (n-1) newGen
    in  (value:restOfList, finalGen)
```

Again, a recursive definition. We say that if we want 0 numbers, we just return an empty list and the generator that was given to us. For any other number of random values, we first get one random number and a new generator. That will be the head. Then we say that the tail will be *n - 1* numbers generated with the new generator. Then we return the head and the rest of the list joined and the final generator that we got from getting the *n - 1* random numbers.

What if we want a random value in some sort of range? All the random integers so far were outrageously big or small. What if we want to to throw a die? Well, we use `randomR` for that purpose. It has a type of `randomR :: (RandomGen g, Random a) :: (a, a) -> g -> (a, g)`, meaning that it's kind of like `random`, only it takes as its first parameter a pair of values that set the lower and upper bounds and the final value produced will be within those bounds.

```
ghci> randomR (1,6) (mkStdGen 359353)
(6,1494289578 40692)
ghci> randomR (1,6) (mkStdGen 35935335)
(3,1250031057 40692)
```

There's also `randomRs`, which produces a stream of random values within our defined ranges. Check this out:

```
ghci> take 10 $ randomRs ('a','z') (mkStdGen 3) :: [Char]
"ndkxbvmomg"
```

Nice, looks like a super secret password or something.

You may be asking yourself, what does this section have to do with I/O anyway? We haven't done anything concerning I/O so far. Well, so far we've always made our random number generator manually by making it with some arbitrary integer. The problem is, if we do that in our real programs, they will always return the same random numbers, which is no good for us. That's why `System.Random` offers the `getStdGen` I/O action, which has a type of `IO StdGen`. When your program starts, it asks the system for a good random number generator and stores that in a so called global generator. `getStdGen` fetches you that global random generator when you bind it to something.

Here's a simple program that generates a random string.

```haskell
import System.Random

main = do
    gen <- getStdGen
    putStr $ take 20 (randomRs ('a','z') gen)
```

```
$ runhaskell random_string.hs
pybphhzzhuepknbykxhe
$ runhaskell random_string.hs
eiqgcxykivpudlsvvjpg
$ runhaskell random_string.hs
nzdceoconysdgcyqjruo
$ runhaskell random_string.hs
```

```
bakzhnnuzrkgvesqplrx
```

Be careful though, just performing `getStdGen` twice will ask the system for the same global generator twice. If you do this:

```haskell
import System.Random

main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen)
    gen2 <- getStdGen
    putStr $ take 20 (randomRs ('a','z') gen2)
```

you will get the same string printed out twice! One way to get two different strings of length 20 is to set up an infinite stream and then take the first 20 characters and print them out in one line and then take the second set of 20 characters and print them out in the second line. For this, we can use the `splitAt` function from `Data.List`, which splits a list at some index and returns a tuple that has the first part as the first component and the second part as the second component.

```haskell
import System.Random
import Data.List

main = do
    gen <- getStdGen
    let randomChars = randomRs ('a','z') gen
        (first20, rest) = splitAt 20 randomChars
        (second20,_) = splitAt 20 rest
    putStrLn first20
    putStr second20
```

Another way is to use the newStdGen action, which splits our current random generator into two generators. It updates the global random generator with one of them and encapsulates the other as its result.

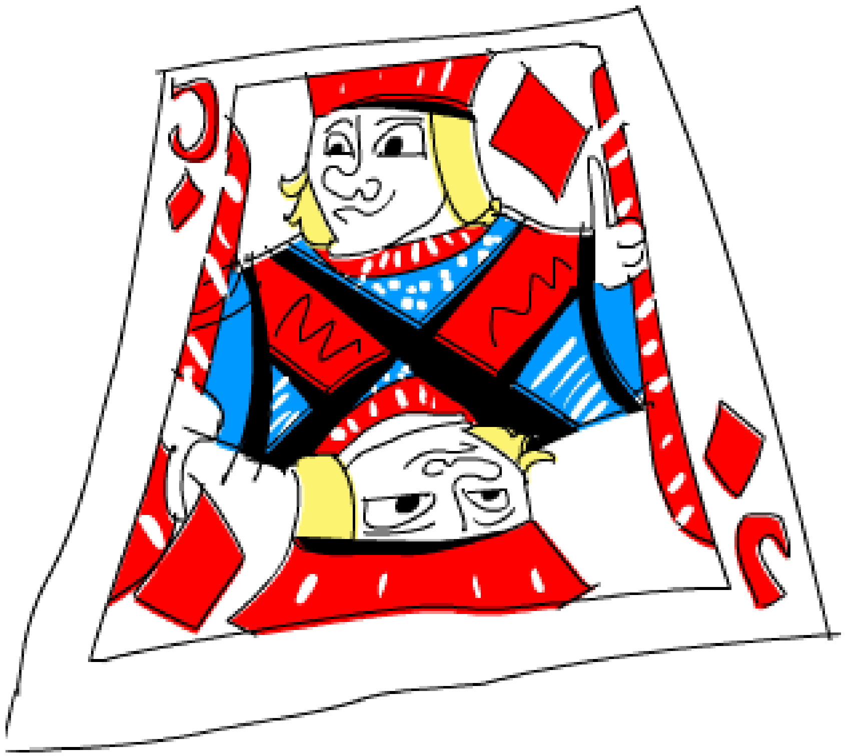
```
import System.Random

main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen)
    gen' <- newStdGen
    putStr $ take 20 (randomRs ('a','z') gen')
```

Not only do we get a new random generator when we bind **newStdGen** to something, the global one gets updated as well, so if we do **getStdGen** again and bind it to something, we'll get a generator that's not the same as **gen**.

Here's a little program that will make the user guess which number it's thinking of.

```
import System.Random
import Control.Monad(when)

main = do
    gen <- getStdGen
    askForNumber gen
```

```haskell
askForNumber :: StdGen -> IO ()
askForNumber gen = do
    let (randNumber, newGen) = randomR (1,10) gen :: (Int, StdGen)
    putStr "Which number in the range from 1 to 10 am I thinking of? "
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was " ++ show randNumber
        askForNumber newGen
```



We make a function `askForNumber`, which takes a random number generator and returns an I/O action that will prompt the user for a number and tell him if he guessed it right. In that function, we first generate a random number and a new generator based on the generator that we got as a parameter and call them `randNumber` and `newGen`. Let's say that the number generated was `7`. Then we tell the user to guess which number we're thinking of. We perform `getLine` and bind its result to `numberString`. When the user enters `7`, `numberString` becomes `"7"`. Next, we use `when` to check if the string the user entered is an empty string. If it is, an empty I/O action of `return ()` is performed, which effectively ends the program. If it isn't, the action consisting of that *do* block right there gets performed. We use `read` on `numberString` to convert it to a number, so `number` is now `7`.

Excuse me! If the user gives us some input here that `read` can't read (like `"haha"`), our program will crash with an ugly error message. If you don't want your program to crash on erronous input, use `reads`, which returns an empty list when it fails to read a string. When it succeeds, it returns a singleton list with a tuple that has our desired value as one component and a string with what it didn't consume as the other.

We check if the number that we entered is equal to the one generated randomly and give the user the appropriate message. And then we call `askForNumber` recursively, only this time with the new generator that we got, which gives us an I/O action that's just like the one we performed, only it depends on a different generator and we perform it.

`main` consists of just getting a random generator from the system and calling `askForNumber` with it to get the initial action.

Here's our program in action!

```
$ runhaskell guess_the_number.hs
Which number in the range from 1 to 10 am I thinking of? 4
Sorry, it was 3
Which number in the range from 1 to 10 am I thinking of? 10
You are correct!
Which number in the range from 1 to 10 am I thinking of? 2
Sorry, it was 4
Which number in the range from 1 to 10 am I thinking of? 5
Sorry, it was 10
```

```
Which number in the range from 1 to 10 am I thinking of?
```

Another way to make this same program is like this:

```haskell
import System.Random
import Control.Monad(when)

main = do
    gen <- getStdGen
    let (randNumber, _) = randomR (1,10) gen :: (Int, StdGen)
    putStr "Which number in the range from 1 to 10 am I thinking of? "
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was " ++ show randNumber
        newStdGen
        main
```

It's very similar to the previous version, only instead of making a function that takes a generator and then calls itself recursively with the new updated generator, we do all the work in `main`. After telling the user whether they were correct in their guess or not, we update the global generator and then call `main` again. Both approaches are valid but I like the first one more since it does less stuff in `main` and also provides us with a function that we can reuse easily.

Bytestrings

Lists are a cool and useful data structure. So far, we've used them pretty much everywhere. There are a multitude of functions that operate on them and Haskell's laziness allows us to exchange the for and while loops of other languages for filtering and mapping over lists, because evaluation will only happen once it really needs to, so things like infinite lists (and even infinite lists of infinite lists!) are no problem for us. That's why lists can also be used to represent streams, either when reading from the standard input or when reading from files. We can just open a file and read it as a string, even though it will only be accessed when the need arises.

However, processing files as strings has one drawback: it tends to be slow. As you know, `String` is a type synonym for `[Char]`. `Char`s don't have a fixed size, because it takes several bytes to represent a character from, say, Unicode. Furthemore, lists are really lazy. If you have a list like `[1,2,3,4]`, it will be evaluated only when completely necessary. So the whole list is sort of a promise of a list. Remember that `[1,2,3,4]` is syntactic sugar for `1:2:3:4:[]`. When the first element of the list is forcibly evaluated (say by printing it), the rest of the list `2:3:4:[]` is still just a promise of a list, and so on. So you can think of lists as promises that the next element will be delivered once it really has to and along with it, the promise of the element after it. It doesn't take a big mental leap to conclude that processing a simple list of numbers as a series of promises might not be the most efficient thing in the world.

That overhead doesn't bother us so much most of the time, but it turns out to be a liability when reading big files and manipulating them. That's why Haskell has **bytestrings**. Bytestrings are sort of like lists, only each element is one byte (or 8 bits) in size. The way they handle laziness is also different.

Bytestrings come in two flavors: strict and lazy ones. Strict bytestrings reside in `Data.ByteString` and they do away with the laziness completely. There are no promises involved; a strict bytestring represents a series of bytes in an array. You can't have things like infinite strict bytestrings. If you evaluate the first byte of a strict bytestring, you have to evaluate it whole. The upside is that there's less overhead because there are no thunks (the technical term for *promise*) involved. The downside is that they're likely to fill your memory up faster because they're read into memory at once.

The other variety of bytestrings resides in `Data.ByteString.Lazy`. They're lazy, but not quite as lazy as lists. Like we said before, there are as many thunks in a list as there are elements. That's what makes them kind of slow for some purposes. Lazy bytestrings take a different approach — they are stored in chunks (not to be confused with thunks!), each chunk has a size of 64K. So if you evaluate a byte in a lazy bytestring (by printing it or something), the first 64K will be evaluated. After that, it's just a promise for the rest of the chunks. Lazy bytestrings are kind of like lists of strict bytestrings with a size of 64K. When you process a file with lazy bytestrings, it will be read chunk by chunk. This is cool because it won't cause the memory usage to skyrocket and the 64K probably fits neatly into your CPU's L2 cache.

If you look through the documentation for `Data.ByteString.Lazy`, you'll see that it has a lot of functions that have the same names as the ones from `Data.List`, only the type signatures have `ByteString` instead of `[a]` and `Word8` instead of `a` in them. The functions with the same names mostly act the same as the ones that work on lists. Because the names are the same, we're going to do a qualified import in a script and then load that script into GHCI to play with bytestrings.

```haskell
import qualified Data.ByteString.Lazy as B
import qualified Data.ByteString as S
```

`B` has lazy bytestring types and functions, whereas `S` has strict ones. We'll mostly be using the lazy version.

The function `pack` has the type signature `pack :: [Word8] -> ByteString`. What that means is that it takes a list of bytes of type `Word8` and returns a `ByteString`. You can think of it as taking a list, which is lazy, and making it less lazy, so that it's lazy only at 64K intervals.

What's the deal with that `Word8` type? Well, it's like `Int`, only that it has a much smaller range, namely 0-255. It represents an 8-bit number. And just like `Int`, it's in the `Num` typeclass. For instance, we know that the value `5` is polymorphic in that it can act like any numeral type. Well, it can also take the type of `Word8`.

```
ghci> B.pack [99,97,110]
Chunk "can" Empty
ghci> B.pack [98..120]
Chunk "bcdefghijklmnopqrstuvwx" Empty
```

As you can see, you usually don't have to worry about the `Word8` too much, because the type system can makes the numbers choose that type. If you try to use a big number, like `336` as a `Word8`, it will just wrap around to `80`.

We packed only a handful of values into a `ByteString`, so they fit inside one chunk. The `Empty` is like the `[]` for lists.

`unpack` is the inverse function of `pack`. It takes a bytestring and turns it into a list of bytes.

`fromChunks` takes a list of strict bytestrings and converts it to a lazy bytestring. `toChunks` takes a lazy bytestring and converts it to a list of strict ones.

```
ghci> B.fromChunks [S.pack [40,41,42], S.pack [43,44,45], S.pack [46,47,48]]
Chunk "()*" (Chunk "+,-" (Chunk "./0" Empty))
```

This is good if you have a lot of small strict bytestrings and you want to process them efficiently without joining them into one big strict bytestring in memory first.

The bytestring version of `:` is called `cons` It takes a byte and a bytestring and puts the byte at the beginning. It's lazy though, so it will make a new chunk even if the first chunk in the bytestring isn't full. That's why it's better to use the strict version of `cons`, `cons'` if you're going to be inserting a lot of bytes at the beginning of a bytestring.

```
ghci> B.cons 85 $ B.pack [80,81,82,84]
Chunk "U" (Chunk "PQRT" Empty)
ghci> B.cons' 85 $ B.pack [80,81,82,84]
Chunk "UPQRT" Empty
ghci> foldr B.cons B.empty [50..60]
Chunk "2" (Chunk "3" (Chunk "4" (Chunk "5" (Chunk "6" (Chunk "7" (Chunk "8" (Chunk "9" (Chunk ":"
Empty))))))))))
ghci> foldr B.cons' B.empty [50..60]
```

```
Chunk "23456789:;<" Empty
```

As you can see `empty` makes an empty bytestring. See the difference between `cons` and `cons'`? With the `foldr`, we started with an empty bytestring and then went over the list of numbers from the right, adding each number to the beginning of the bytestring. When we used `cons`, we ended up with one chunk for every byte, which kind of defeats the purpose.

Otherwise, the bytestring modules have a load of functions that are analogous to those in `Data.List`, including, but not limited to, `head`, `tail`, `init`, `null`, `length`, `map`, `reverse`, `foldl`, `foldr`, `concat`, `takeWhile`, `filter`, etc.

It also has functions that have the same name and behave the same as some functions found in `System.IO`, only `String`s are replaced with `ByteString`s. For instance, the `readFile` function in `System.IO` has a type of `readFile :: FilePath -> IO String`, while the `readFile` from the bytestring modules has a type of `readFile :: FilePath -> IO ByteString`. Watch out, if you're using strict bytestrings and you attempt to read a file, it will read it into memory at once! With lazy bytestrings, it will read it into neat chunks.

Let's make a simple program that takes two filenames as command-line arguments and copies the first file into the second file. Note that `System.Directory` already has a function called `copyFile`, but we're going to implement our own file copying function and program anyway.

```
import System.Environment
import qualified Data.ByteString.Lazy as B

main = do
```

```
    (fileName1:fileName2:_) <- getArgs
    copyFile fileName1 fileName2

copyFile :: FilePath -> FilePath -> IO ()
copyFile source dest = do
    contents <- B.readFile source
    B.writeFile dest contents
```

We make our own function that takes two `FilePath`s (remember, `FilePath` is just a synonym for `String`) and returns an I/O action that will copy one file into another using bytestring. In the `main` function, we just get the arguments and call our function with them to get the I/O action, which is then performed.

```
$ runhaskell bytestringcopy.hs something.txt ../../something.txt
```

Notice that a program that doesn't use bytestrings could look just like this, the only difference is that we used `B.readFile` and `B.writeFile` instead of `readFile` and `writeFile`. Many times, you can convert a program that uses normal strings to a program that uses bytestrings by just doing the necessary imports and then putting the qualified module names in front of some functions. Sometimes, you have to convert functions that you wrote to work on strings so that they work on bytestrings, but that's not hard.

Whenever you need better performance in a program that reads a lot of data into strings, give bytestrings a try, chances are you'll get some good performance boosts with very little effort on your part. I usually write programs by using normal strings and then convert them to use bytestrings if the performance is not satisfactory.

Exceptions

All languages have procedures, functions, and pieces of code that might fail in some way. That's just a fact of life. Different languages have different ways of handling those failures. In C, we usually use some abnormal return value (like `-1` or a null pointer) to indicate that what a function returned shouldn't be treated like a normal value. Java and C#, on the other hand, tend to use exceptions to handle failure. When an exception is thrown, the control flow jumps to some code that we've defined that does some cleanup and then maybe re-throws the exception so that some other error handling code can take care of some other stuff.

Haskell has a very good type system. Algebraic data types allow for types like `Maybe` and `Either` and we can use values of those types to represent results that may be there or not. In C, returning, say, `-1` on failure is completely a matter of convention. It only has special meaning to humans. If we're not careful, we might treat these abnormal values as ordinary ones and then they can cause havoc and dismay in our code. Haskell's type system gives us some much-needed safety in that aspect. A function `a -> Maybe b` clearly indicates that it it may produce a `b` wrapped in `Just` or that it may return `Nothing`. The type is different from just plain `a -> b` and if we try to use those two functions interchangeably, the compiler will complain at us.

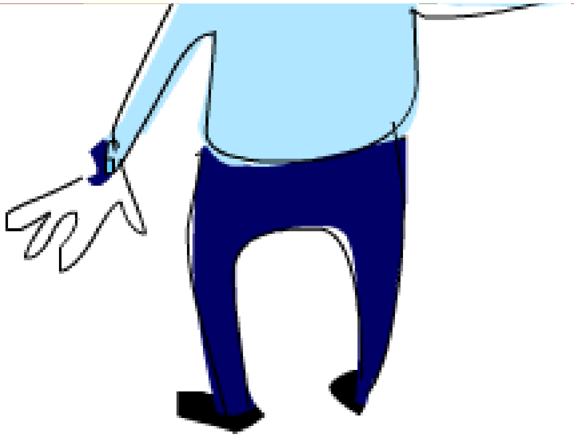
Despite having expressive types that support failed computations, Haskell still has support for exceptions, because they make more sense in I/O contexts. A lot of things can go wrong when dealing with the outside world because it is so unreliable. For instance, when opening a file, a bunch of things can go wrong. The file might be locked, it might not be there at all or the hard disk drive or something might not be there at all. So it's good to be able to jump to some error handling part of our code when such an error occurs.

Okay, so I/O code (i.e. impure code) can throw exceptions. It makes sense. But what about pure code? Well, it can throw exceptions too. Think about the `div` and `head` functions. They have types of `(Integral a) => a -> a -> a` and `[a] -> a`, respectively. No `Maybe` or `Either` in their return type and yet they can both fail! `div` explodes in your face if you try to divide by zero and `head` throws a tantrum when you give it an empty list.

```
ghci> 4 `div` 0
*** Exception: divide by zero
ghci> head []
*** Exception: Prelude.head: empty list
```

Pure code can throw exceptions, but it they can only be caught in the I/O part of our code (when we're inside a *do* block that goes into `main`). That's because you don't know when (or if) anything will be evaluated in pure code, because it is lazy and doesn't have a well-defined order of execution, whereas I/O code does.

Earlier, we talked about how we should spend as little time as possible in the I/O part of our program. The logic of our program should reside mostly within our pure



functions, because their results are dependant only on the parameters that the functions are called with. When dealing with pure functions, you only have to think about what a function returns, because it can't do anything else. This makes your life easier. Even though doing some logic in I/O is necessary (like opening files and the like), it should preferably be kept to a minimum. Pure functions are lazy by default, which means that we don't know when they will be evaluated and that it really shouldn't matter. However, once pure functions start throwing exceptions, it matters when they are evaluated. That's why we can only catch exceptions thrown from pure functions in the I/O part of our code. And that's bad, because we want to keep the I/O part as small as possible. However, if we don't catch them in the I/O part of our code, our program crashes. The solution? Don't mix exceptions and pure code. Take advantage of Haskell's powerful type system and use types like `Either` and `Maybe` to represent results that may have failed.

That's why we'll just be looking at how to use I/O exceptions for now. I/O exceptions are exceptions that are caused when something goes wrong while we are communicating with the outside world in an I/O action that's part of `main`. For example, we can try opening a file and then it turns out that the file has been deleted or something. Take a look at this program that opens a file whose name is given to it as a command line argument and tells us how many lines the file has.

```haskell
import System.Environment
import System.IO

main = do (fileName:_) <- getArgs
          contents <- readFile fileName
          putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"
```

A very simple program. We perform the `getArgs` I/O action and bind the first string in the list that it yields to `fileName`. Then we call the contents of the file with that name `contents`. Lastly, we apply `lines` to those contents to get a list of lines and then we get the length of that list and give it to `show` to get a string representation of that number. It works as expected, but what happens when we give it the name of a file that doesn't exist?

```
$ runhaskell linecount.hs i_dont_exist.txt
linecount.hs: i_dont_exist.txt: openFile: does not exist (No such file or directory)
```

Aha, we get an error from GHC, telling us that the file does not exist. Our program crashes. What if we wanted to print out a nicer message if the file doesn't exist? One way to do that is to check if the file exists before trying to open it by using the `doesFileExist` function from `System.Directory`.

```haskell
import System.Environment
import System.IO
import System.Directory

main = do (fileName:_) <- getArgs
          fileExists <- doesFileExist fileName
          if fileExists
              then do contents <- readFile fileName
                      putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"
              else do putStrLn "The file doesn't exist!"
```

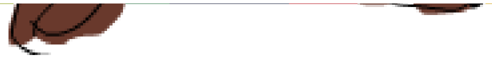
We did `fileExists <- doesFileExist fileName` because `doesFileExist` has a type of `doesFileExist :: FilePath -> IO Bool`, which means that it returns an I/O action that has as its result a boolean value which tells us if the file exists. We can't just use `doesFileExist` in an *if* expression directly.

Another solution here would be to use exceptions. It's perfectly acceptable to use them in this context. A file not existing is an exception that arises from I/O, so catching it in I/O is fine and dandy.

To deal with this by using exceptions, we're going to take advantage of the `catch` function from `System.IO.Error`. Its type is `catch :: IO a -> (IOError -> IO a) -> IO a`. It takes two parameters. The first one is an I/O action. For instance, it could be an I/O action that tries to open a file. The second one is the so-called handler. If the first I/O action passed to `catch` throws an I/O exception, that exception gets passed to the handler, which then decides what to do. So the final result is an I/O action that will either act the same as the first parameter or it will do what the handler tells it if the first I/O action throws an exception.

If you're familiar with *try-catch* blocks in languages like Java or Python, the `catch` function is similar to them. The first parameter is the thing to try, kind of like the stuff in the *try* block in other, imperative languages. The second parameter is the handler that takes an exception, just like most *catch* blocks take exceptions that you can then examine to see what happened. The handler is invoked if an exception is thrown.

The handler takes a value of type `IOError`, which is a value that



signifies that an I/O exception occurred. It also carries information regarding the type of the exception that was thrown. How this type is implemented depends on the implementation of the language itself, which means that we can't inspect values of the type `IOError` by pattern matching against them, just like we can't pattern match against values of type `IO something`. We can use a bunch of useful predicates to find out stuff about values of type `IOError` as we'll learn in a second.

So let's put our new friend `catch` to use!

```haskell
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e = putStrLn "Whoops, had some trouble!"
```

First of all, you'll see that put backticks around it so that we can use it as an infix function, because it takes two parameters. Using it as an infix function makes it more readable. So ``toTry `catch` handler`` is the same as `catch toTry handler`, which fits well with its type. `toTry` is the I/O action that we try to carry out and `handler` is the function that takes an `IOError` and returns an action to be carried out in case of an exception.

Let's give this a go:

```
$ runhaskell count_lines.hs i_exist.txt
The file has 3 lines!

$ runhaskell count_lines.hs i_dont_exist.txt
Whoops, had some trouble!
```

In the handler, we didn't check to see what kind of `IOError` we got. We just say `"Whoops, had some trouble!"` for any kind of error. Just catching all types of exceptions in one handler is bad practice in Haskell just like it is in most other languages. What if some other exception happens that we don't want to catch, like us interrupting the program or something? That's why we're going to do the same thing that's usually done in other languages as well: we'll check to see what kind of exception we got. If it's the kind of exception we're waiting to catch, we do our stuff. If it's not, we throw that exception back into the wild. Let's modify our program to catch only the exceptions caused by a file not existing.

```
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_) <- getArgs
```

```haskell
            contents <- readFile fileName
            putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e = putStrLn "The file doesn't exist!"
    | otherwise = ioError e
```

Everything stays the same except the handler, which we modified to only catch a certain group of I/O exceptions. Here we used two new functions from `System.IO.Error` — `isDoesNotExistError` and `ioError`. `isDoesNotExistError` is a predicate over `IOError`s, which means that it's a function that takes an `IOError` and returns a `True` or `False`, meaning it has a type of `isDoesNotExistError :: IOError -> Bool`. We use it on the exception that gets passed to our handler to see if it's an error caused by a file not existing. We use guard syntax here, but we could have also used an *if else*. If it's not caused by a file not existing, we re-throw the exception that was passed by the handler with the `ioError` function. It has a type of `ioError :: IOException -> IO a`, so it takes an `IOError` and produces an I/O action that will throw it. The I/O action has a type of `IO a`, because it never actually yields a result, so it can act as `IO anything`.

So the exception thrown in the `toTry` I/O action that we glued together with a *do* block isn't caused by a file existing, `` toTry `catch` handler `` will catch that and then re-throw it. Pretty cool, huh?

There are several predicates that act on `IOError` and if a guard doesn't evaluate to `True`, evaluation falls through to the next guard. The predicates that act on `IOError` are:

- `isAlreadyExistsError`

- `isDoesNotExistError`
- `isAlreadyInUseError`
- `isFullError`
- `isEOFError`
- `isIllegalOperation`
- `isPermissionError`
- `isUserError`

Most of these are pretty self-explanatory. `isUserError` evaluates to `True` when we use the function `userError` to make the exception, which is used for making exceptions from our code and equipping them with a string. For instance, you can do `ioError $ userError "remote computer unplugged!"`, although It's prefered you use types like `Either` and `Maybe` to express possible failure instead of throwing exceptions yourself with `userError`.

So you could have a handler that looks something like this:

```
handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e = putStrLn "The file doesn't exist!"
    | isFullError e = freeSomeSpace
    | isIllegalOperation e = notifyCops
    | otherwise = ioError e
```

Where `notifyCops` and `freeSomeSpace` are some I/O actions that you define. Be sure to re-throw exceptions if they don't match any of your criteria, otherwise you're causing your program to fail silently in some cases where it shouldn't.

`System.IO.Error` also exports functions that enable us to ask our exceptions for some attributes, like what the handle of the file that caused the error is, or what the filename is. These start with `ioe` and you can see a full list of them in the documentation. Say we want to print the filename that caused our error. We can't print the `fileName` that we got from `getArgs`, because only the `IOError` is passed to the handler and the handler doesn't know about anything else. A function depends only on the parameters it was called with. That's why we can use the `ioeGetFileName` function, which has a type of `ioeGetFileName :: IOError -> Maybe FilePath`. It takes an `IOError` as a parameter and maybe returns a `FilePath` (which is just a type synonym for `String`, remember, so it's kind of the same thing). Basically, what it does is it extracts the file path from the `IOError`, if it can. Let's modify our program to print out the file path that's responsible for the exception occurring.

```haskell
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e =
        case ioeGetFileName e of Just path -
> putStrLn $ "Whoops! File does not exist at: " ++ path
                                 Nothing -
> putStrLn "Whoops! File does not exist at unknown location!"
    | otherwise = ioError e
```

In the guard where `isDoesNotExistError` is `True`, we used a *case* expression to call `ioeGetFileName` with `e` and then pattern match against the `Maybe` value that it returned. Using *case* expressions is commonly used when you want to pattern match against something without bringing in a new function.

You don't have to use one handler to `catch` exceptions in your whole I/O part. You can just cover certain parts of your I/O code with `catch` or you can cover several of them with `catch` and use different handlers for them, like so:

```
main = do toTry `catch` handler1
          thenTryThis `catch` handler2
          launchRockets
```

Here, `toTry` uses `handler1` as the handler and `thenTryThis` uses `handler2`. `launchRockets` isn't a parameter to `catch`, so whichever exceptions it might throw will likely crash our program, unless `launchRockets` uses `catch` internally to handle its own exceptions. Of course `toTry`, `thenTryThis` and `launchRockets` are I/O actions that have been glued together using *do* syntax and hypothetically defined somewhere else. This is kind of similar to *try-catch* blocks of other languages, where you can surround your whole program in a single *try-catch* or you can use a more fine-grained approach and use different ones in different parts of your code to control what kind of error handling happens where.

Now you know how to deal with I/O exceptions! Throwing exceptions from pure code and dealing with them hasn't been covered here, mainly because, like we said, Haskell offers much better ways to indicate errors than reverting to I/O to catch them. Even when glueing together I/O actions that might fail, I prefer to have their type be something like `IO (Either a b)`, meaning that they're normal I/O actions but the result that they yield when performed is of type `Either a b`, meaning it's either `Left a` or `Right b`.

← Making Our Own Types and Typeclasses   Table of contents   Functionally Solving Problems →