Hello, class!
How are you? Good? That's good.

Oh, me? I'm fine as well. Thanks for asking!

Hey! Know what would make doing IO and such easier?

... well?

Aren't you going to answer?
Oh right! You can't. Sorry. I'll just assume you would have said something like, “Why no, oh great and amazing Earl. What would make doing IO and such easier?"

A new kind of loop!

Remember that the general idea of loops is to continuously repeat some steps until you decide that (big surprise) you no longer wish to loop!
In Java and such, we typically used `while` loops, which had a condition testing if you should continue.

Our new Prolog loop will behave differently in a few ways, so pay attention!
The special command is `repeat`.
When you tell it to `repeat`, what you're *really* doing is setting up an infinite branch-point.
What this means is that, if part of the remainder of the subgoals fails (that is, if some term after the repeat can't be unified), if the inference engine needs to backtrack all the way back to the `repeat` line, it will treat it as having found a new branch for execution.

Note that this is both very powerful and very dangerous.
First and foremost, it relies on the assumption that the remainder of the subgoals after the `repeat` line **will all solve eventually**. Otherwise, it will get stuck in an infinite loop and eventually run out of stack space.

However, it's also powerful in that it allows for far easier control over execution.

Obviously, this is one of those extra-logical tools.

Refer to the `looping.pro` file.
The `simpleread` predicate includes a very simple demonstration of how to use repeat.

```
?- simpleread.
Type 'quit.' to finish; any other single word to display
|: hey.
hey
|: class.
class
|: quit.
quit
true.
```

Next, try looking at `readonline`. It doesn't loop yet, but rather simply tries to read a single line from a text file. It's what we're going to slowly build our next attempt on.
(Note: You need to get `inputfile.txt` as well, and put it in the same folder)

```
?- readoneline.
This is the first line!
true.
```

Huzzah! We can read... one line of text! Just like we already could! Yaaay!

Now, let's look at a predicate to read/process all of the lines of text. You have the `looping.pro`, but let's remind ourselves of what it looks like.
(Sorry if this spoils the surprise, but it isn't going to work yet)

```
readALLthelines:-     %Broom not included
     current_input(OldStream),
     see('inputfile.txt'),
     seeing(NewStream),
     repeat, %'repeat' comes after we've opened our stream!
     read_line_to_codes(NewStream,ASCIIList),
     name(Line,ASCIIList),
     write(Line),nl, %This is the end of the current read.
     %something clever could go here.
     see(OldStream).
```

So, how well does it work? (You saw the spoiler, didn't you?)

```
?- readALLthelines.
This is the first line!
true ;
And this is the SECOND LINE! WOOO!
true ;
And, just so you know, there'll effectively be a blank line...
true ;
right after this line!
true ;
ERROR: name/2: Type error: `list' expected, found `end_of_file'
```

Wow. That sucked, eh?
So then, two problems:

1. A `repeat` loop only actually, like, *loops* if some term after it fails.
2. We haven't considered how to handle the end-of-file case.

We could try to fix our code intelligently, by carefully analyzing our requirements, and then properly forming a reasonable algorithm that suitably meets our needs.

Or... we could just mash the keyboard until we make some progress. I like that idea. Let's do that!

```
readALLthelinesTotally:-
     current_input(OldStream),
     see('inputfile.txt'),
     seeing(NewStream),
     repeat,
     read_line_to_codes(NewStream,ASCIIList),
     name(Line,ASCIIList),
     write(Line),nl,
     fail, %This 'fail' GUARANTEES that it will loop! ... forever.
     see(OldStream).
```

Surely, this will work! Right?
(Spoiler alert: No. No, it won't. But it'll get us closer!)

```
?- readALLthelinesTotally.
This is the first line!
And this is the SECOND LINE! WOOO!
And, just so you know, there'll effectively be a blank line...
right after this line!
ERROR: name/2: Type error: `list' expected, found `end_of_file'
```

So, this time, you can see that it clearly was able to keep looping until it had processed all of the data! It's still complaining because we still haven't accounted for the end of the file, but we knew we'd have to get to that sooner or later, anyway.

What we really need to do is to treat it as trying to be at the end of the file, and failing otherwise. However, if we fail to be at the end of the line, we still obviously need to do work, because that means we have data to process!

```
readALLthelinesForReal:-
     current_input(OldStream),
     see('inputfile.txt'),
     seeing(NewStream),
     repeat,
     read_line_to_codes(NewStream,ASCIIList),
     (ASCIIList=end_of_file->true;
          (name(Line,ASCIIList),write(Line),nl,fail)),
     seen,
     see(OldStream),!.
```

Howsabout we give this one a try?

```
?- readALLthelinesForReal.
This is the first line!
And this is the SECOND LINE! WOOO!
And, just so you know, there'll effectively be a blank line...
right after this line!
true.
```

Well now, that's much better, isn't it?
Take another look at the code. This time, we decided to use a conditional (hey, if we're already polluting our code with extra-logical terms, why not muddy it up even more?). Basically, if we've hit the end of the file, then we declare that we're good so far. Otherwise, we do whatever work we wish to do on the data we've just read, and then declare that we actually failed.

What this means is that, when we read in the first line, we reform it into a string, print that to the screen, and then declare that we need to 'go back and try it again', which takes us back to the last branch-point (the `repeat`). This leads us to repeatedly read from the file, right up until there's nothing left to read, which then puts us in that desired state of having succeeded in exhausting the file. As such, we're free to continue on (or, in this case, close our stream and then end).

So, now we're good. But, realistically, this is only readable because we aren't really doing that much work in terms of "processing" the data. What if we wanted to address that?

```
readALLthelinesnicely:-
     current_input(OldStream),
     see('inputfile.txt'),
     seeing(NewStream),
     repeat,
     read_line_to_codes(NewStream,ASCIIList),
     (ASCIIList=end_of_file->true;
          process_data(ASCIIList) /*fail could be here instead*/),
     seen,
     see(OldStream),!.

process_data(ASCII):-
     name(Message,ASCII),
     write(Message),nl,
     assert(wisdom(Message)),
     !, %This cut is likely overly cautious
     fail. %This fail could also be handled above instead
```

In this case, just for ha-ha's, we also decided to store our read information in memory, one-line-per-fact.