Characters, Strings and Files
The Character Datatype

Literal Characters
In Java the letter 'a' is denoted with quote marks. Quotes have a different meaning in Scheme, so we need something else. In Scheme the letter 'a' is denoted #\a; the letter 'A' is #\A, the digit '1' is #\1, and so forth.
Converting characters to integers, and vice versa:

(char->integer c) gives the index of character c in the standard table of characters. (char->integer #\a) is 97; (char->integer #\A is 65 and so forth.

Similarly, (integer->char n) returns the nth character in the standard table of characters: (integer->char 97) is #\a
Character Comparison

You can't use < to compare two character values:
(< \a \b) gives an error. Instead you need to use operators such as char<?  char<=? and so forth. If they weren't built in you could easily define these yourself:

(define char <? (lambda ( a b) (< (char->integer a) (char->integer b)))))
Character Predicates

There are some built-in procedures that recognize specific sets of characters:

(char-alphabetic? c)
(char-numeric? c)
(char-whitespace? c)

All of these will give an error if they are called with an argument that is not a character.
The String Datatype

Strings are represented in double-quotes, as in "bob". There are a few built-in string procedures that we will use:

(string ch1 ch2 ch3 ... chn) concatenates the n characters into a string. For example (string #\b #\o #\b) returns the string "bob".

(string) returns the empty string.

(string-ref s n) returns the nth character of string s.

(string-append s1 s2) returns a new string that is the concatenation of s1 and s2.
There are also string comparison operators: string<?, string<=?, etc.
Output in Scheme
There are (print x) and (println x) procedures that output the value of x. The difference between them is that println terminates the current line.
For example
  (begin
    (print "bob")
    (print (+ 16 7))
  )
prints "bob"23
and
  (begin
    (println "bob")
    (print (+ 16 7))
  )
prints
  "bob"
  23
There is also a formatted string facility in Scheme that works similarly to such facilities in Java and Python. In Scheme it works only with formatted print statements (where this notion originated, in C).

(printf string arg1 arg2 ...) 

prints the given string, replacing the first ~field with the value of the first argument, the second ~field with the value of the second argument and so forth.
There are two kinds of \texttt{~}fields: \texttt{~s} prints its argument as the argument itself would be printed (e.g., if the argument is a string the quotation marks around the string are included), while \texttt{~a} does a minimal amount of processing (so quotes around strings are not included). For example

\begin{verbatim}
(printf "Dear \texttt{~s}: please send $\texttt{~s}." "Dad" (+10 10) )
\end{verbatim}

will print

Dear "Dad": please send $20.

while

\begin{verbatim}
(printf "Dear \texttt{~a}: please send $\texttt{~a}." "Dad" (+10 10) )
\end{verbatim}

will print

Dear Dad: please send $20.
printf does not terminate the output line the way println does. To terminate the line include ~% in the output string:

   (printf "one~%two~%three~%)

will print

   one
   two
   three
There are several ways we can get input from the user of a Scheme program:

(read) prompts the user for input, and echoes whatever the user types. However, this returns the input interpreted as a Scheme value. If you type blah, the input is the symbol 'blah. If you type (1 2 3) the input is a list with three numbers. If you type "bob" the input is the string "bob". In all of these cases, the input is a valid Scheme expression. However, if you type blah blah blah blah, the value returned is just the single symbol 'blah.
File input

The procedure

(open-input-file <file name as a string>)

if comparable to a file open statement in Java or Python. It
returns a value that we can treat as an input source, so we
might say something like

(define input (open-input-file "calculation.txt"))

We can then read the input one character at a time with

(read-char input)

which returns the next character of the input.

The procedure (eof-object? x) when given a value returned
by read-char will return #t if we have reached the end of
the file.
String input

The procedure

(open-input-string s)

is an alternative way to get input, this time from a string. For example, we might say

(define input (open-input-string "3+4*5"))

and then read input one character at a time with

(read-char input)

(eof-object? x) when given a value returned by read-char will say if we have reached the end of the input string.
Here, for example, is a short program that reads and reassembles an input string one character at a time:

```
(define input (open-input-string "3*4+5"))

(define myReader (lambda ()
    (let ([c (read-char input)])
        (if (not (eof-object? c))
            (if (not (eof-object? c))
                (string-append (string c) (myReader))
                (string)))),
        (string)))))

(f)
```