```
(ns map-filter-reduce.core
  (:gen-class))

(defn -main
  "I don't do a whole lot ... yet."
  [& args]
  (println "Hello, World!"))
```

```
#'map-filter-reduce.core/-main
```

```

```

```
; Given a list of numbers.
(range 10)
```

```
(0 1 2 3 4 5 6 7 8 9)
```

```
; We can quickly map over all of them and add one to
each element.
(map inc (range 10))
```

```
(1 2 3 4 5 6 7 8 9 10)
```

```
; Mappings work great for when all elements in a
list can be transformed independently.
; Here we can transform all numbers between 97 and
122 to their ascii related characters.

(map char (range 97 122))
```

```
(\a \b \c \d \e \f \g \h \i \j \k \l \m \n \o \p \q \r
\s \t \u \v \w \x \y)
```

```
; But something seems off there. I really like the
little quotes around strings.
; Maybe we will just map str over the last line.

(map str (map char (range 97 122)))
```

```
("a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
"n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y")
```

```
; The above does the job but it feels messy. We are
iterating over the list twice.
; Once to pass all the elements of (range 97 122) to
char and again for str.
; Why not iterate just once?

(map #(str (char %)) (range 97 122))
```

```
("a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
"n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y")
```

```
; So what just happened up there? What is this hash
nonsense and why is there a percent sign?
; What we just saw was an anonymous function or
"lambda" and they are what make programming in
clojure pleasant!

; The "#" is a macro that declares whatever is
within the following paranthesis to be a function.
; The "%" represents that anonymous function or
lamba's input.
; Try to figure out what the mapping below will
evaluate to before you evaluate it.

(map #(= % %) (range 10))
```

```
(true true true true true true true true true true)
```

```
; Now try writing your own anonymous lambda using
the "#" macro and "%" below!
```

```
; Try writing another one here!
```

```
; Okay now that you may have the hang of how basic
mappings work. Let's quickly go over filter and
reduce.
```

```
; Filter is a function that take a conditional
function that returns true or false and a list or
vector as arguments.
; Filter then returns everything in the list where
the conditional function is true.

(filter even? (range 10))
```

```
(0 2 4 6 8)
```

```
; Filter can also take an anonymous function as its
first argument.

(filter #(= (mod % 2) 0) (range 10))
```

```
(0 2 4 6 8)
```

```
(filter #(= (mod % 2) 1) (range 10))
```

```
(1 3 5 7 9)
```

```
; Remove is exactly like filter except that it
returns everything is the list where the conditional
function is false.
(remove #(= (mod % 2) 0) (range 10))
```

```
(1 3 5 7 9)
```

```
(remove #(= (mod % 2) 1) (range 10))
```

```
(0 2 4 6 8)
```

```
; Reduce is a function that takes a function and a
list as arguments and returns a single value.
; Here we can sum all numbers from 0 - 10 using
reduce.
(reduce + (range 11))
```

```
55
```

```
; Here we can use reduce to calculate the product of
a series of numbers.
(reduce * (range 1 10))
```

```
362880
```

```
; Reduce works by taking the first two elements in a
list
(reduce #(* (+ %1 %1) %2) (range 2 5))
```

```
96
```

```
; I hope this has been enough of a crash course for
everyone.
; I strongly suggest checking out chapters 3 & 4 of
; "Clojure for the Brave and True"
; For a couple of great explanations of map and
reduce with diagrams.

; Now I know I said that I would show you all how to
write a multithreaded program in clojure so here it
is!

(time (remove number? (doall (pmap #(reduce + %)
(repeatedly 100000 #(vec (range 11)))))))
```

```
"Elapsed time: 272.72425 msecs"
```

```
()
```

```
; The above function is almost exactly the same as
the function except for one letter.
; The letter "p" in pmap signifies that it is a
parallel mapping function in clojure.
; All it takes to add a little parralelism in
clojure to most of your code is the letter "p"!
; Below is the same function as above without the p.
There are calls here that many of you may have not
seen.

;(doall) forces eager evaluation.
; time is a macro the prints how long it takes a
function to return.

(time (remove number? (doall (map #(reduce + %)
(repeatedly 100000 #(vec (range 11)))))))
```

```
"Elapsed time: 76.508963 msecs"
```

```
()
```

```
; Now you may have noticed that the multithreaded
pmap call took longer to return in this instance
than the map call.
; Why is that? Shouldn't more threads be faster?

; More threads can be faster.
; However there is an initial overhead cost in
segmenting parts of the list to be evaluated in
different threads.
; pmap should only be really used for titanic jobs
on many cored computers.

; If you run into computational time problems that
can not be solved with optimization remember that
Hampshire has a cluster for Evo Comp.

; Just ask Lee if you can run your code on the
cluster and make sure to use pmap to get the most
out of one cluster node!
```