We’ve seen how to use `define`, `let`, `let*`, and `letrec` to assign values to variables.

But we’ve never changed the value associated with a variable after we initially set it.

This is a good way to write functions!

We can focus on each function’s inputs and outputs, and everything of importance happens in the body of the function definition.

The only side effect we use is printing, which won’t affect the computation.

This makes it much easier to find errors in a function; just look at the body of the function.
However, there are some problems that are going to require us to change the value of a variable or to change the value of a part of a data structure.

This is called *destructive programming* because it will overwrite the previous value and can't be undone.

To highlight their destructive nature, the names for the functions we use will end with an exclamation mark.

You should follow this convention when you write a destructive function too!

A function that changes an existing value is called a **mutator**.

The general-purpose mutator is the **set!** special form:

```
> (define name "anna")
> name
"anna"
> (set! name "banana")
> name
"banana"
```

*set!* only works on symbols that have already been defined:

```
> (set! banana 42)
Error
```

Remember that if you **set!** a symbol to a new value, the old value is lost permanently. However, if you make a copy of a symbol's definition, that should remain, e.g.,

```
> (define backup (void))
> (define name "He Who Must Not Be Named")
> (set! backup name)
> (define name "Voldemort!")
> name
"Voldemort!"
> backup
"He Who Must Not Be Named"
> (set! name backup)
> name
"He Who Must Not Be Named"
```
If you use `set!` inside, say, a `let`, it will affect the “nearest” symbol of that name. Try:

```scheme
> (define x 'xxx)
> (define y 'yyy)
> (let ((x 42))
  (set! x 'new-x)
  (set! y 'new-y)
  (list x y))
(new-x new-y)
> (list x y)
(xxx new-y)
```

Vectors are another built-in data structure.

In many ways, a vector is like a list, but they’re much more efficient because they provide random access.

This means that the time it takes to fetch a random element of a vector does not depend on where it is in the vector.

Contrast this with lists, where finding the 99,999th element requires walking through 99,998 elements before arriving at the desired element.

Vectors are stored in contiguous blocks of memory, so they can take advantage of the computer’s random access memory (RAM).

This means when you create a vector, you need to know how big it must be.

Resizing vectors is something we want to avoid because it requires reallocating another block of memory.

Thus, when using vectors, we tend to allocate a fixed-size vector. Then, later on, we will destructively alter the contents of the vector as the need arises!
**Constructor 1: `make-vector`**

*Inputs:* length of the desired vector

  initial value of each element (optional; default is 0)

*Output:* An n-element vector of initial values

> `(make-vector 0)`

#()

> `(make-vector 5)`

#(0 0 0 0 0)

> `(make-vector 5 "banana")`

#("banana" "banana" "banana" "banana" "banana")

**Constructor 2: `vector`**

This procedure takes a bunch of objects and forms a vector containing them, similar to the `list` constructor:

> `(vector 'a 'b 'c 'd 'e 'f 'g)`

#(a b c d e f g)

Note that Scheme displays a vector prefixed by a # to distinguish it from a list.

**Constructor 3: `#` prefix operator**

Just as we can use `quote` to construct lists of unevaluated expressions, we can use `#` to construct vectors of unevaluated expressions:

> `#(a b c d e f g)`

#(a b c d e f g)

**Beware:** When you use the `#` operator to create a vector, the result is *immutable*, i.e., it cannot be changed!

Immutable vectors are generally used for storing data that is only going to be read (i.e., looked up) and never needs to change while the program is running.

Since a vector has a fixed length, we can instantly check its length:

```scheme
(vector-length (make-vector 1000000))
```

1000000

Remember that using `length` to check the number of elements in a list is slow; it needs to go through every element in the list, one at a time, counting them up.
**Accessor**: vector-ref

\[
\text{vector-ref}(\text{vector name} \langle \text{position} \rangle)
\]

```lisp
> (define vec (make-vector 10))
> (vector-ref vec 4))
0
```

**Type checker predicate**: vector?

```lisp
> (vector? (make-vector 10))
#t
> (vector? '(1 2 3))
#f
> (vector? #(1 2 3))
#t
```

**Mutator**: vector-set!

\[
\text{vector-set!}(\text{vector name} \langle \text{position} \rangle \langle \text{value} \rangle)
\]

```lisp
> (define vec (make-vector 10))
> (vector-set! vec 4 321)
> vec
#(0 0 0 321 0 0 0 0 0 0)
```

```lisp
;; FETCH-RANDOM-ELEMENT
;; ---------------------
;; INPUTS: VEC, a vector
;; OUTPUT: One of the elements of VEC, chosen at random
(define fetch-random-element
  (lambda (vec)
    (let ((r (random (vector-length vec))))
      (printf "R = ~A~%" r)
      ;; Access the Rth element
      (vector-ref vec r)))))

> (fetch-random-element #(a b c d e f g))
R = 0
a
> (fetch-random-element #(a b c d e f g))
R = 3
d
```
;; MODIFY-RANDOM-ELEMENT!
;; ----------------------
;; INPUT: VECKY, a vector
;; OUTPUT: The changed vector
;; SIDE EFFECT: Changes the value of a
;; random selected element of VECKY to X
(define modify-random-element!
  (lambda (vec)
    (let (; R: random index into VEC
         (r (random (vector-length vec))))
      (printf "R = ~A~%" r)
        ;; Modify the Rth element
        (vector-set! vec r 'X))
      ;; output the changed vector
      vec))

(tester '(modify-random-element! (vector 1 2 3 4 5)))
(tester '(modify-random-element! (vector 'a 'b 'c #t #f ())))
(define myveck (vector 32 8 16 5))
(tester '(modify-random-element! myveck))
(tester 'myveck)

> (define-struct person (name age))
> (define harry (make-person "Harry" 10))
> (person-name harry)
"Harry"
> (set-person-name! harry "Henry")
> (person-name harry)
"Henry"
There are also constructions that let us loop without using recursion or map. One of these is while:

```
(define counting-up
  (lambda (from to)
    (while (<= from to)
      (printf "~A " from)
      (set! from (+ from 1)))))
```

```
> (counting-up 1 10)
1 2 3 4 5 6 7 8 9 10
```

Compare the iterative and recursive versions of the same function:

```
(define sum-all-iter
  (lambda (n)
    (let ((counter 0)
           (acc 0))
      (while (<= counter n)
        (set! acc (+ acc counter))
        (set! counter (+ counter 1)))
      ;; After the WHILE loop: the accumulator has the answer acc))

> (sum-all-iter 5)
15
> (sum-all-iter 100)
5050
> (sum-all-iter 3)
6
```

```
(define sum-all-recur
  (lambda (n counter acc)
    (cond ;; Recursive Case
      ((<= counter n)
        (sum-all-recur n (+ counter 1) (+ acc counter)))
      ;; Base Case
      (else acc)))

> (sum-all-recur 3 0 0)
6
```

Acknowledgments

This lecture incorporates material from:

Simon Ellis
Luke Hunsberger