Without set!, the operands were being used in the mathematical sense like an alias for a value.

Since set! allows the notion of changing an entity's value, it introduces the notion that the entity has an existence independent of its value.

This notion is not suited for the mathematical nature of functional programming, but it is quite well suited for modelling real world objects whose properties also change over time.

We ordinarily view the world as consisting of independent objects, each of which has a state that changes over time.
We can characterize an object's state by one or more *state variables*, which among them maintain enough information about history to determine the object's current behavior.

For example:-

- balance (state variable) in a bank account (object) or
- weight(state variable) of a human being (object).

These objects are rarely completely independent.

Each may influence the states of others through interactions, which serve to couple the state variables of one object to those of other objects.

For example, employer's bank account might interact with the employee's bank account.

This view of a system composed of separate objects is most useful when the state variables of the system can be grouped into closely coupled subsystems that are only loosely coupled to other subsystems.

For such a model to be modular, it should be decomposed into **computational objects that model the actual objects** in the system.

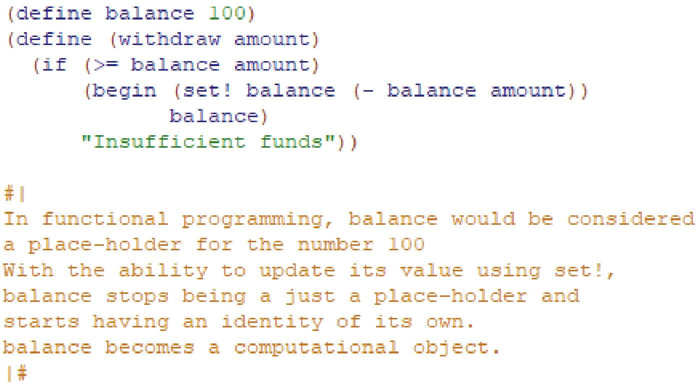
Each computational object must have its own local state variables describing the actual object's state.

The states of objects in the system being modeled change over time.

Therefore, the state variables of the corresponding computational objects must also change.

If we wish to model state variables by ordinary symbolic names in the programming language, then the language must provide an assignment operator (like set!) to enable us to change the value associated with a name.

The intention to model real-world objects which undergo change as opposed to mathematical objects which do not undergo change is a valid reason to add set! to the language.



```
(define balance 100)
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds"))

#|
In functional programming, balance would be considered
a place-holder for the number 100
With the ability to update its value using set!,
balance stops being a just a place-holder and
starts having an identity of its own.
balance becomes a computational object.
|#
```

```
; In the aforementioned piece of code, one can
; circumvent the use of withdraw procedure and
; directly update the balance using set!

(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds"))))

; In new-withdraw, we make balance object internal
; to the procedure and thereby avoid the problem
; which withdraw procedure faced.
```

The variable ‘balance’ is said to be encapsulated within the new-withdraw procedure.

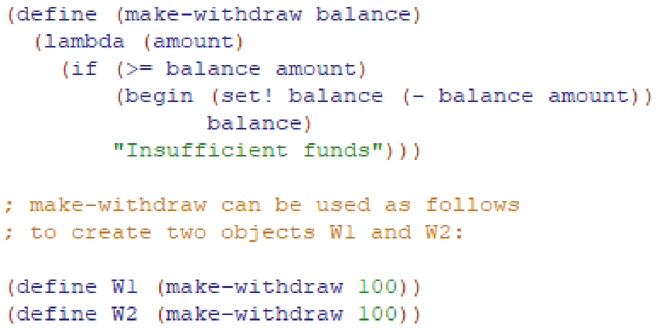
Encapsulation reflects the general system-design principle known as the hiding principle:

This procedure—returned as the result of evaluating the let expression—is new-withdraw, which behaves in precisely the same way as withdraw but whose variable balance is not accessible by any other procedure.

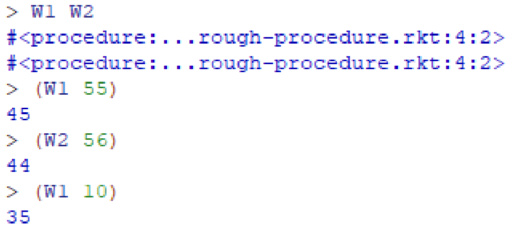
One can make a system more modular and robust by protecting parts of the system from each other; that is, by providing information access only to those parts of the system that have a "need to know".

In this instance only the new-withdraw procedure (which is also a computational object) has the need to know the balance in order to be able to update it.

Combining set! with local variables is the general programming technique we will use for constructing computational objects with local state.



```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))

; make-withdraw can be used as follows
; to create two objects W1 and W2:

(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))
```

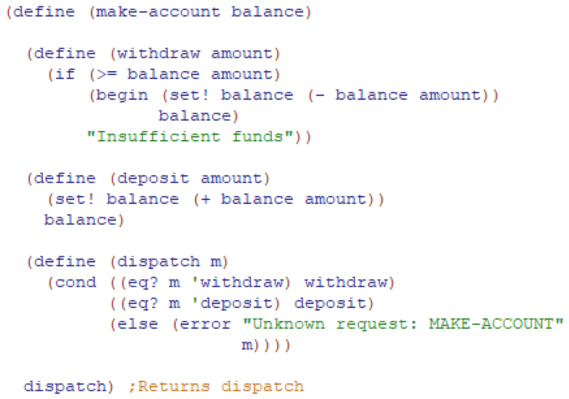


```
> W1 W2
#<procedure:...rough-procedure.rkt:4:2>
#<procedure:...rough-procedure.rkt:4:2>
> (W1 55)
45
> (W2 56)
44
> (W1 10)
35
```

Scheme procedures are first-class objects in the language.

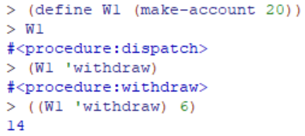
W1 and W2 are completely independent objects, each with its own local state variable balance.

Withdrawals from one object do not affect the balance of the other object.



```scheme
(define (make-account balance)

  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))

  (define (deposit amount)
    (set! balance (+ balance amount))
   balance)

  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request: MAKE-ACCOUNT"
                       m))))

  dispatch) ;Returns dispatch
```

One can go step further and have multiple procedures to change a local state variable. In case of multiple procedures, one would need a default procedure to access the object.
In this case, dispatch is the procedure which redirects us to the procedures we wish to access.

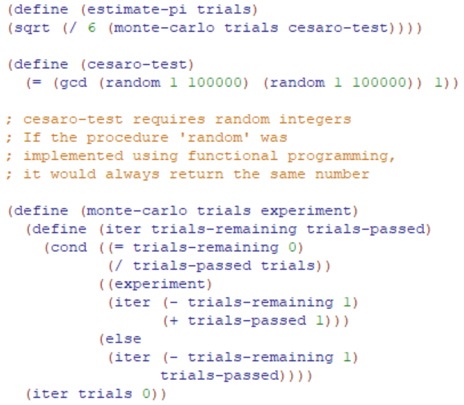


```
> (define W1 (make-account 20))
> W1
#<procedure:dispatch>
> (W1 'withdraw)
#<procedure:withdraw>
> ((W1 'withdraw) 6)
14
```

Each call to make-account sets up an *environment* with a local state variable ‘balance’.

Within this environment, make-account defines procedures deposit and withdraw that access balance and an additional procedure dispatch that takes a “message” as input and returns one of the two local procedures.

The dispatch procedure itself is returned as the value that represents the bank-account object. This reflects the message-passing style of programming which we are using in conjunction with the ability to modify local variables.



```
(define (estimate-pi trials)
(sqrt (/ 6 (monte-carlo trials cesaro-test))))

(define (cesaro-test)
  (= (gcd (random 1 100000) (random 1 100000)) 1))

; cesaro-test requires random integers
; If the procedure 'random' was
; implemented using functional programming,
; it would always return the same number

(define (monte-carlo trials experiment)
  (define (iter trials-remaining trials-passed)
    (cond ((= trials-remaining 0)
           (/ trials-passed trials))
          ((experiment)
           (iter (- trials-remaining 1)
                 (+ trials-passed 1)))
          (else
           (iter (- trials-remaining 1)
                 trials-passed))))
  (iter trials 0))
```

Cesaro's technique estimates π using the fact that
$\frac{6}{\pi^2} =$ the probability that two integers chosen at random have gcd 1.

The expression (random 1 100000) does not always return the same output as one would expect in functional programming

General phenomenon illustrated by the Monte Carlo example:
From the point of view of one part of a complex process, the other parts appear to change with time.

They have hidden time-varying local state.

If we wish to write computer programs whose structure reflects this decomposition, we make computational objects (such as bank accounts and random-number generators) whose behavior changes with time.

We model *state* with local state variables, and
we model the *changes of state* with assignments to those variables.