Topic 16
Assignment and Local State

Section 3.1

Program Design

- So far we have learned about basic elements from which programs are made
  - Primitive procedures and primitive data combined to form compound entities
  - Abstraction (data)
- Here turn to organizational principles for overall design of a program

Two prominent organizational strategies:

- **Object oriented** – program collection of objects whose behaviors may change over time
- **Streams** of information that flow in a system

The state of an object

- Objects have properties that change over time, e.g., snowballs, cars, people
- The **state of an object is the collection of the values of all of its properties at a given moment in time**

Representing objects

- To represent an object, we must represent its state
- The measurable properties of an object can be represented by **local state variables**
- **To make the values of local state variables change over time, an assignment operator is needed, but this will make program behavior harder to predict**

Bank account example

```
(bank-statement) --> 1000
(withdraw 50) --> 950
(withdraw 50) --> 900
(withdraw 990) → "insufficient funds"
(withdraw 50) --> 850
```

`withdraw` **is not a mathematical function!**

Notice that different values are returned for the same function at different times

An implementation

```
; initialize the balance in the account
(define balance 1000)

; takes an amount, if there is enough balance in the account to
; cover the amount, then return the new balance (after withdraw
; is made).  Otherwise, return "insufficient funds"
(define (withdraw amount)
  (if (>= balance amount)
      (begin
        (set! balance
              (- balance amount))
        balance)
      "insufficient funds"))
```

An implementation

```
; initialize the balance in the account
(define balance 1000)

; takes an amount, if there is enough balance in the account to
; cover the amount, then return the new balance (after withdraw
; is made).  Otherwise, return "insufficient funds"
(define (withdraw amount)
  (if (>= balance amount)
      (begin
        (set! balance
              (- balance amount))
        balance)
      "insufficient funds"))
```


An implementation

```
; initialize the balance in the account
(define balance 1000)

; takes an amount, if there is enough balance in the account to
; cover the amount, then return the new balance (after withdraw
; is made).  Otherwise, return "insufficient funds"
(define (withdraw amount)
  (if (>= balance amount)
      (begin
        (set! Balance
              (- balance amount))
        balance)
      "insufficient funds"))
```

set! Is a special form that takes a symbol & an expression. set! changes the value of the symbol so that its value is the value of the expression

An implementation

```
; initialize the balance in the account
(define balance 1000)

; takes an amount, if there is enough balance in the account to
; cover the amount, then return the new balance (after withdraw
; is made).  Otherwise, return "insufficient funds"
(define (withdraw amount)
  (if (>= balance amount)
      (begin
        (set! Balance
              (- balance amount))
        balance)
      "insufficient funds"))
```


An implementation

```
; initialize the balance in the account
(define balance 1000)

; takes an amount, if there is enough balance in the account to
; cover the amount, then return the new balance (after withdraw
; is made).  Otherwise, return "insufficient funds"
(define (withdraw amount)
  (if (>= balance amount)
      (begin
        (set! Balance
              (- balance amount))
        balance)
      "insufficient funds"))
```

begin is a special form that simply evaluates a sequence of expressions – its value is the value of the last one.

Problems with implementation

- balance is a global variable
- Any procedure can change the variable
- But, it should only be available for change to the withdraw procedure.
- Can we do that?
- Can we use the function with more than one account?

Withdrawing from more than one fund

```
; make-withdraw takes a balance as an argument it returns
; a function that expects an amount to be withdrawn as its
; argument -- resets the balance accordingly. Think of this
; function as establishing a balance in an account – and
; returning a procedure that implements withdraws
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance
                     (- balance amount))
               balance)
        "insufficient funds")))
```

A history of withdrawals

```
(define checking-withdraw (make-withdraw 100))
(define savings-withdraw (make-withdraw 500))
> (checking-withdraw 90)
10
> (savings-withdraw 300)
200
> (checking-withdraw 5)
5
> (checking-withdraw 5)
0
> (checking-withdraw 5)
"insufficient funds"
> (savings-withdraw 5)
195
> (savings-withdraw 5)
190
```


More realistic bank accounts

```
; make-account takes a balance and returns
; a procedure which is able to dispatch
; calls to withdraw or deposit both
; of which takes an amount and either withdraws
; or deposits the amount from the balance
(define (make-account balance)
```


(the two features)

```
; takes an amount and sets balance to balance minus
 ; the amount if balance is high enough
(define (withdraw amount)
  (if (>= balance amount)
     (begin (set! balance
                  (- balance amount))
            balance)
     "insufficient funds"))

; takes an amount and increases balance by that amount
(define (deposit amount)
  (set! balance (+ balance amount))
  balance)
```


(the bank teller)

```
  ; dispatches either withdraw or deposit as
  ; appropriate
  (define (dispatch msg)
    (cond ((eq? msg 'withdraw) withdraw)
          ((eq? msg 'deposit) deposit)
          (else
            (error
              "You can't do that here:“
               msg))))
dispatch)
```


Using the new function

```
> (define checking (make-account 100))
> ((checking 'withdraw) 30)
70
> ((checking 'withdraw) 30)
40
> ((checking 'deposit) 400)
440
> ((checking 'withdraw) 30)
410
>
```


Tale of two bank accounts

```
> (define checking (make-account 100))
> (define savings (make-account 500))
> ((checking 'withdraw) 75)
25
> ((savings 'withdraw) 75)
425
> ((checking 'deposit) 5)
30
> ((savings 'withdraw) 125)
300
> ((savings 'withdraw) 500)
"insufficient funds"
> ((savings 'close))
. reference to undefined identifier: error(define acc1 (make-
   account 1000))
```