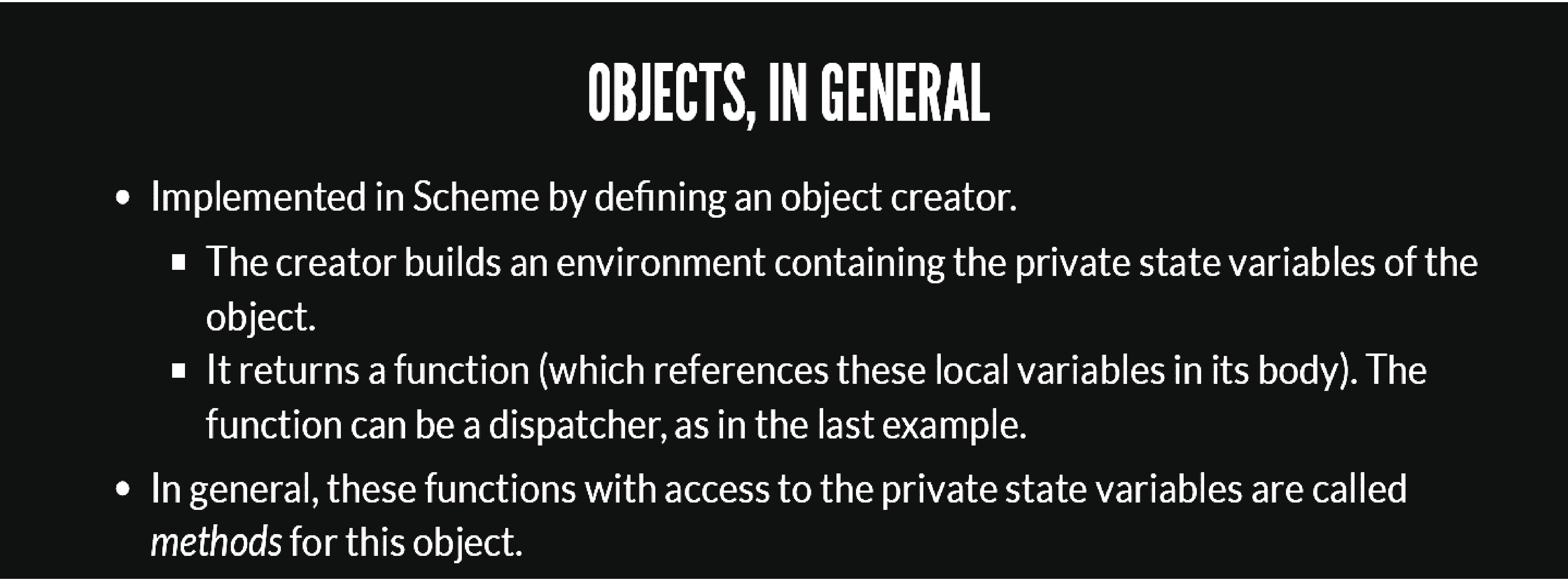
CSE 1729:Principles of Programming

Lecture 21 : Introduction to Pointers

Kaleel Mahmood
Department of Computer Science and Engineering
University of Connecticut

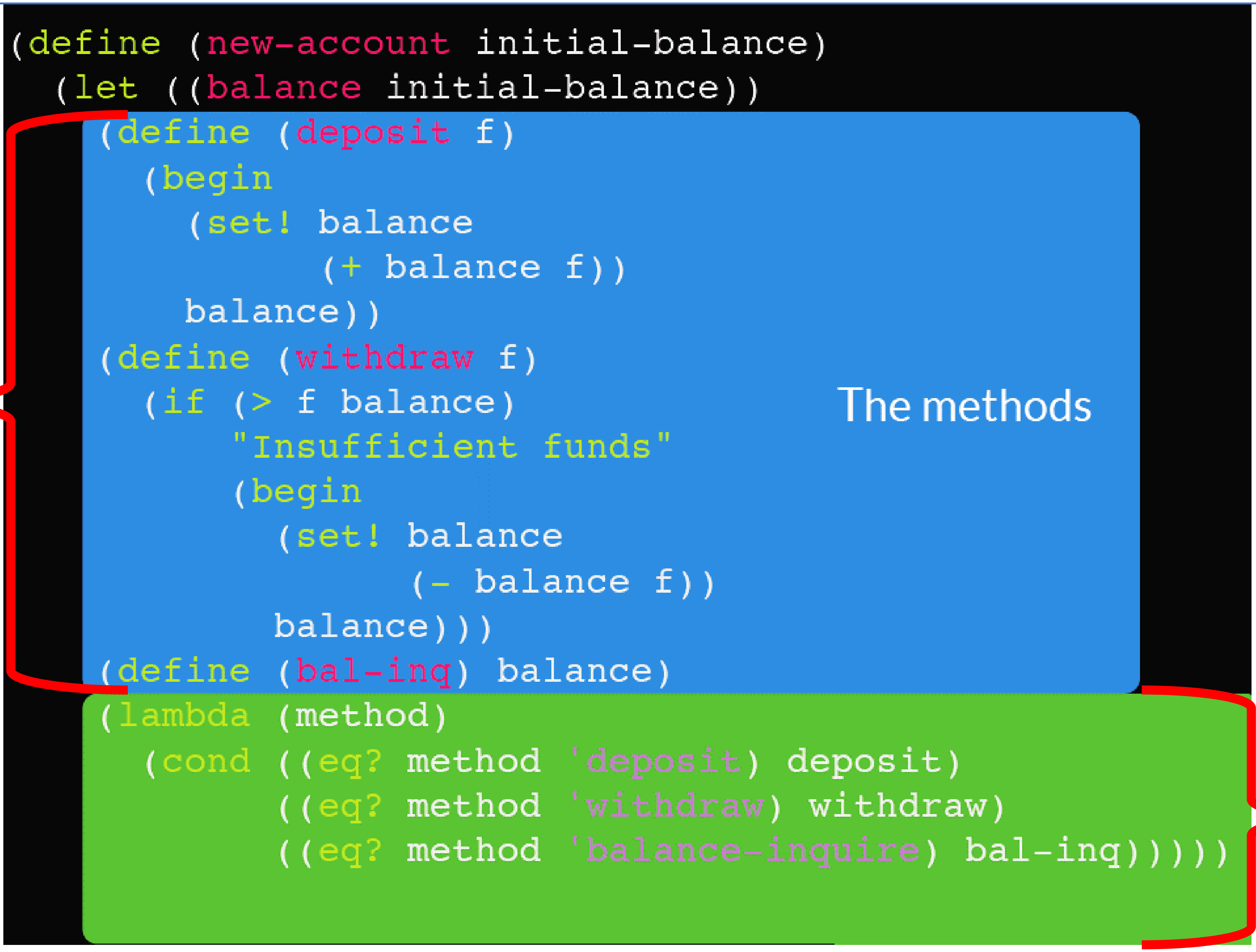
Review from last lecture



OBJECTS, IN GENERAL

- Implemented in Scheme by defining an object creator.
    - The creator builds an environment containing the private state variables of the object.
    - It returns a function (which references these local variables in its body). The function can be a dispatcher, as in the last example.
- In general, these functions with access to the private state variables are called *methods* for this object.

Review from last lecture



```scheme
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (define (deposit f)
      (begin
        (set! balance
              (+ balance f))
        balance))
    (define (withdraw f)
      (if (> f balance)
          "Insufficient funds"
          (begin
            (set! balance
                  (- balance f))
            balance)))
    (define (bal-inq) balance)
    (lambda (method)
      (cond ((eq? method 'deposit) deposit)
            ((eq? method 'withdraw) withdraw)
            ((eq? method 'balance-inquire) bal-inq)))))
```

Public Methods to manipulate private objects

Dispatcher that uses tokens to determine the method call

Stacks

- A container of objects, similar to a stack of coins or Pez dispenser.
- Objects can be inserted at any time.
- Only the top (last placed object) can be removed. Last-in-first out (LIFO).
- Pushing – An object is added to the top of the stack.
- Popping – Removing the top object (the last one added) from the stack.

Figure Source: https://assets.ldscdn.org/09/b6/09b659f83aef1ce638e411d6cedd8cfd92320687/coins.jpeg

A STACK OBJECT THAT IMPLEMENTS THIS ADT

```scheme
(define (make-stack)
  (let ((S '()))
    (define (empty?) (null? S))
    (define (top) (car S))
    (define (pop) (let ((top (car S)))
                    (begin (set! S (cdr S))
                           top)))
    (define (push x) (set! S (cons x S)))
```

Public Methods to manipulate private objects

A STACK OBJECT THAT IMPLEMENTS THIS ADT

```
(define (make-stack)
```

[redacted]

```
   (define (dispatcher method)
     (cond ((eq? method 'top) top)
           ((eq? method 'pop) pop)
           ((eq? method 'push) push)
           ((eq? method 'empty) empty?)))
   dispatcher))
```

Dispatcher that uses tokens to determine the method call

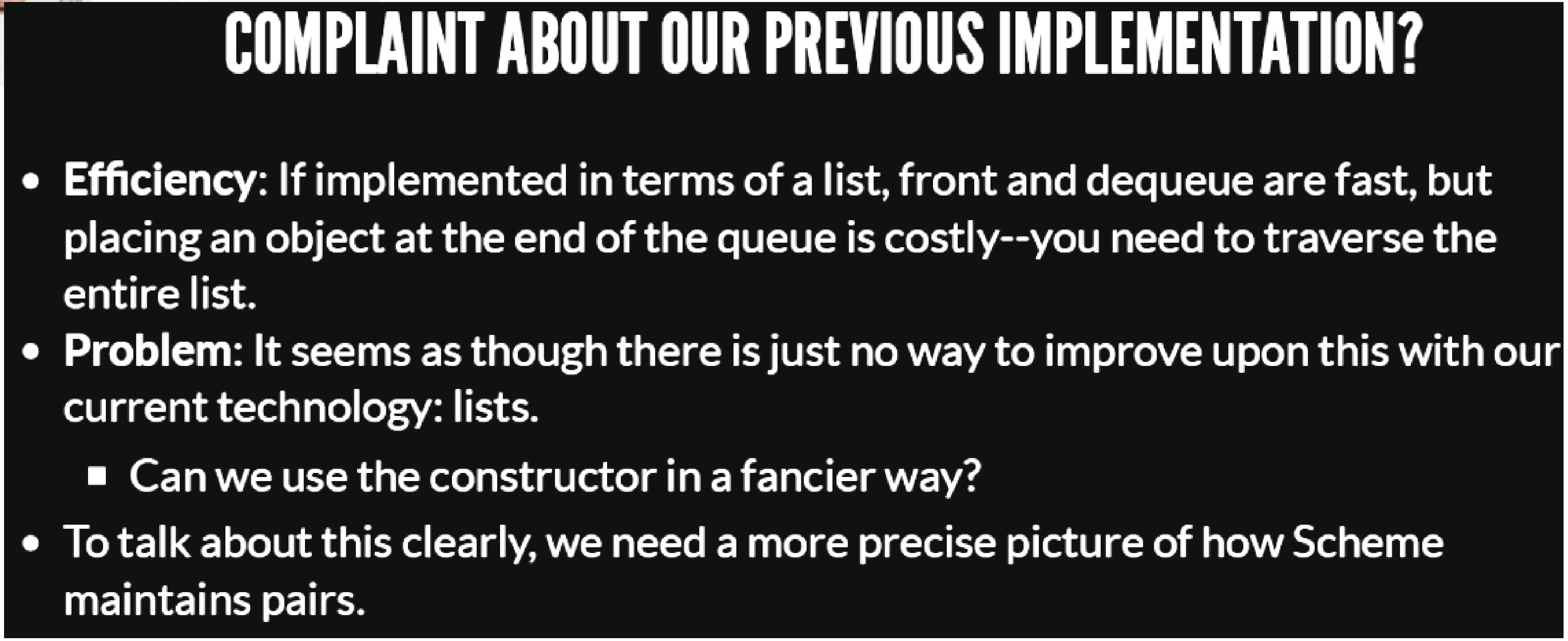
Putting Everything Together...

```scheme
(define (make-stack)
  (let ((S '()))
    (define (empty?) (null? S))
    (define (top) (car S))
    (define (pop) (let ((top (car S)))
                    (begin (set! S (cdr S))
                           top)))
    (define (push x) (set! S (cons x S)))
    (define (dispatcher method)
      (cond ((eq? method 'top) top)
            ((eq? method 'pop) pop)
            ((eq? method 'push) push)
            ((eq? method 'empty) empty?)))
    dispatcher))
```

IMPLEMENTING AN EFFICIENT QUEUE

- Recall the Queue ADT. It models a queue of objects: you can examine the first object, remove it, or enqueue on a new object onto the end.
  - empty?(Q): returns true if the queue is empty, otherwise false
  - front(Q): returns the front object in the queue, without removing it
  - dequeue(Q): removes the top element of the queue.
  - enqueue(x, Q): pushes the element x onto the queue.

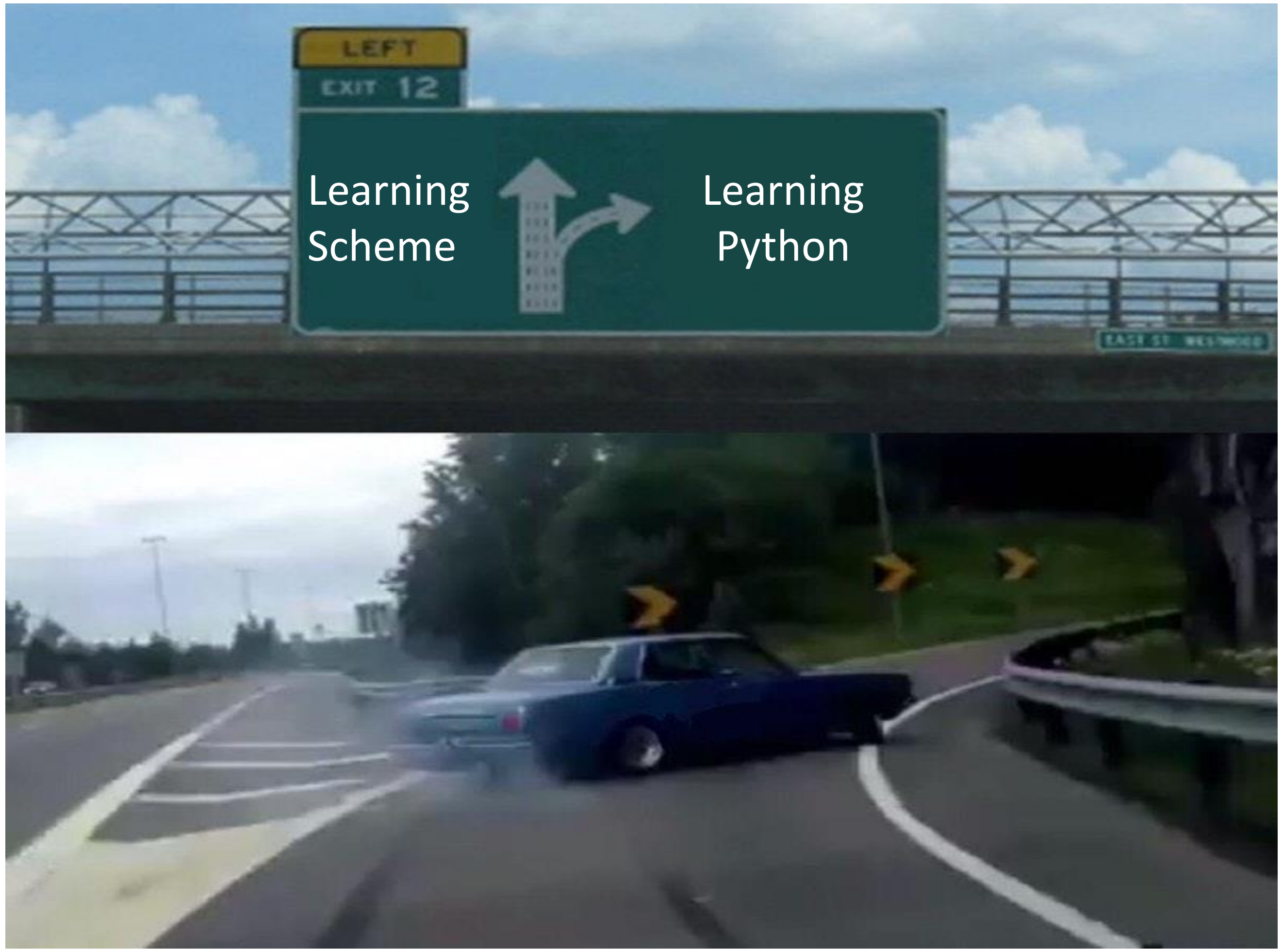
How it sometimes feels to program in Scheme



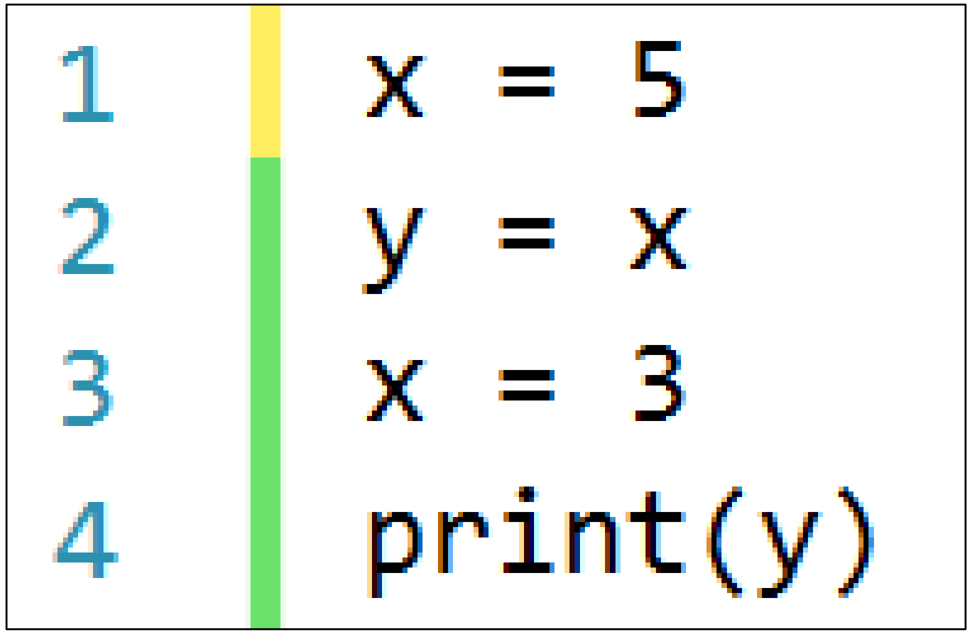
COMPLAINT ABOUT OUR PREVIOUS IMPLEMENTATION?

- **Efficiency**: If implemented in terms of a list, front and dequeue are fast, but placing an object at the end of the queue is costly--you need to traverse the entire list.
- **Problem**: It seems as though there is just no way to improve upon this with our current technology: lists.
  - Can we use the constructor in a fancier way?
- To talk about this clearly, we need a more precise picture of how Scheme maintains pairs.

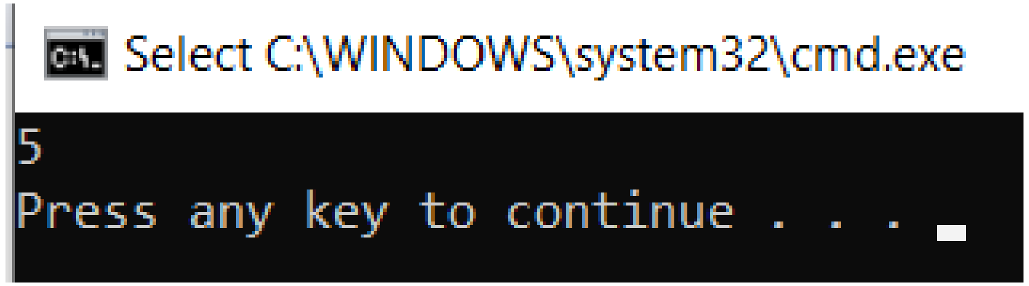
To build a better Queue in any language we need to understand pointers this requires a detour…



Code Example 1: What is the value of y?



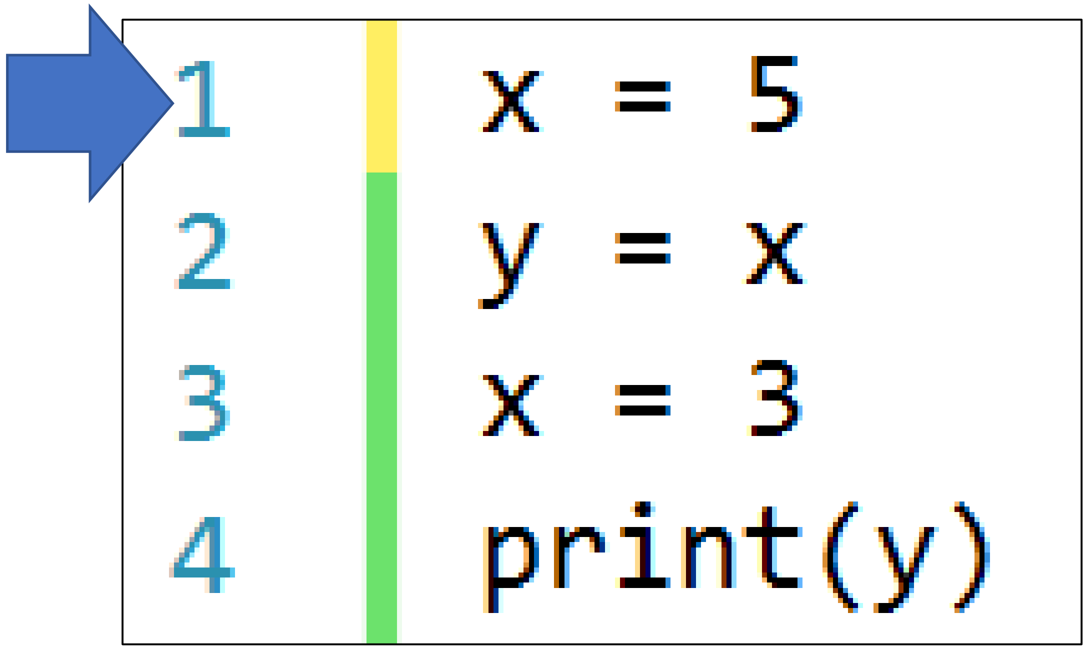
```
x = 5
y = x
x = 3
print(y)
```



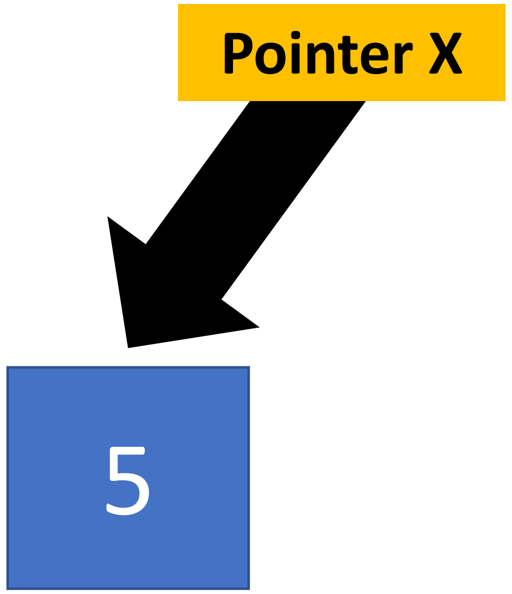
Select C:\WINDOWS\system32\cmd.exe

```
5
Press any key to continue . . .
```

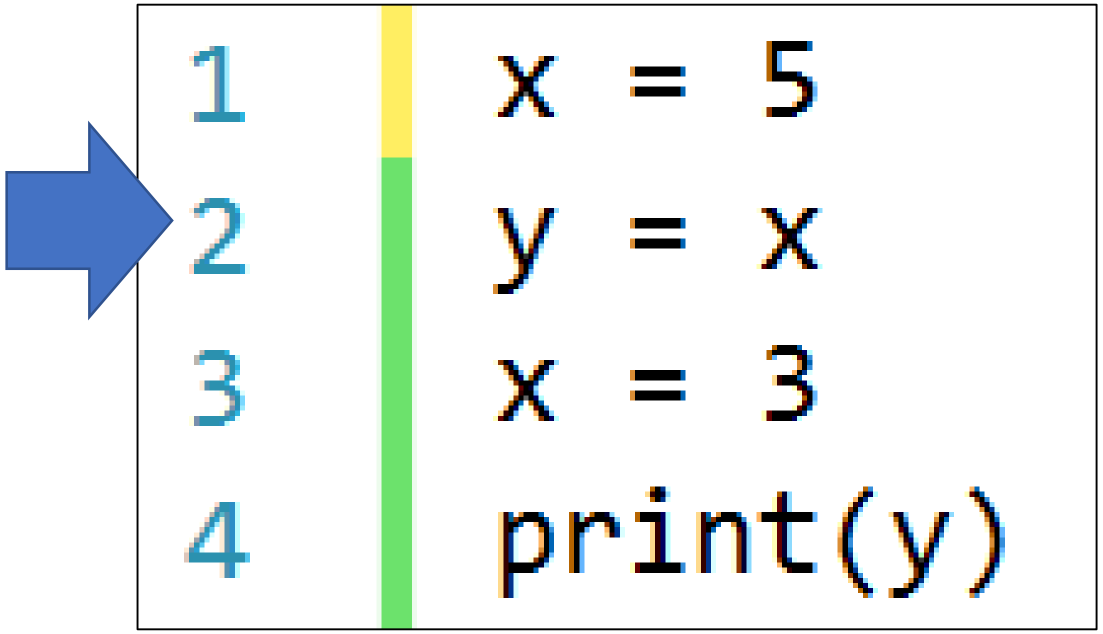
Tracing through the code 1



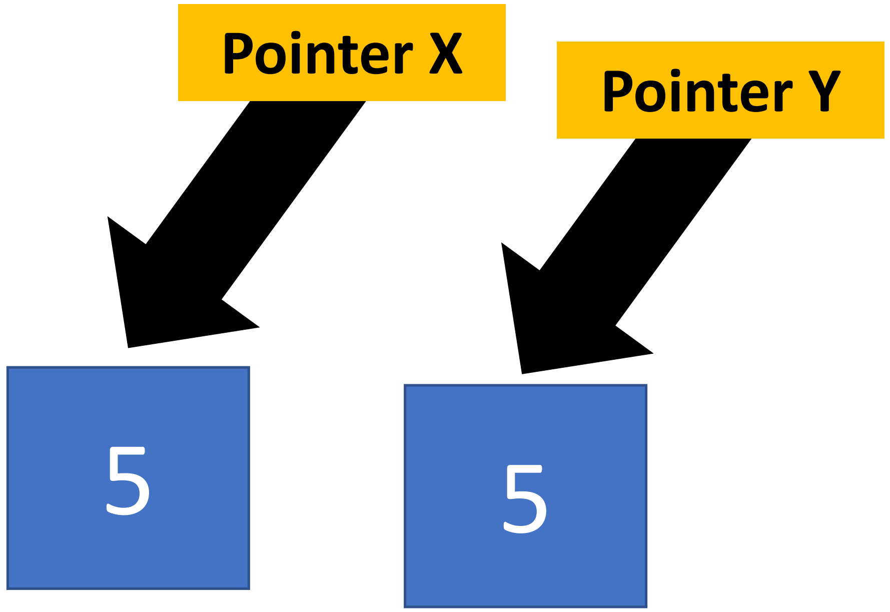
```
x = 5
y = x
x = 3
print(y)
```



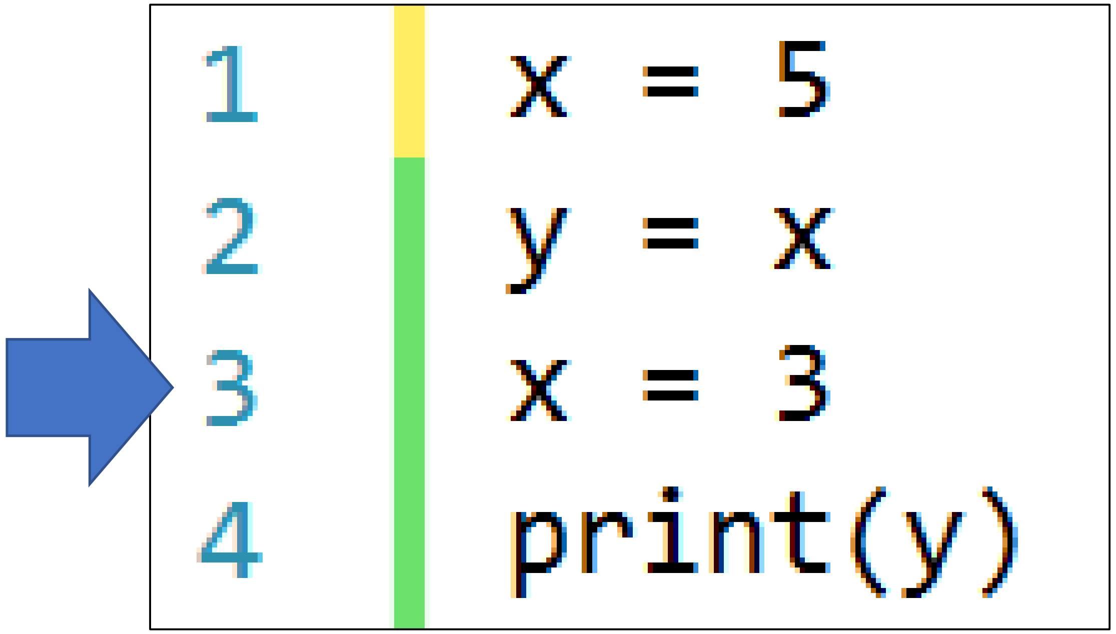
Tracing through the code 2



```
x = 5
y = x
x = 3
print(y)
```

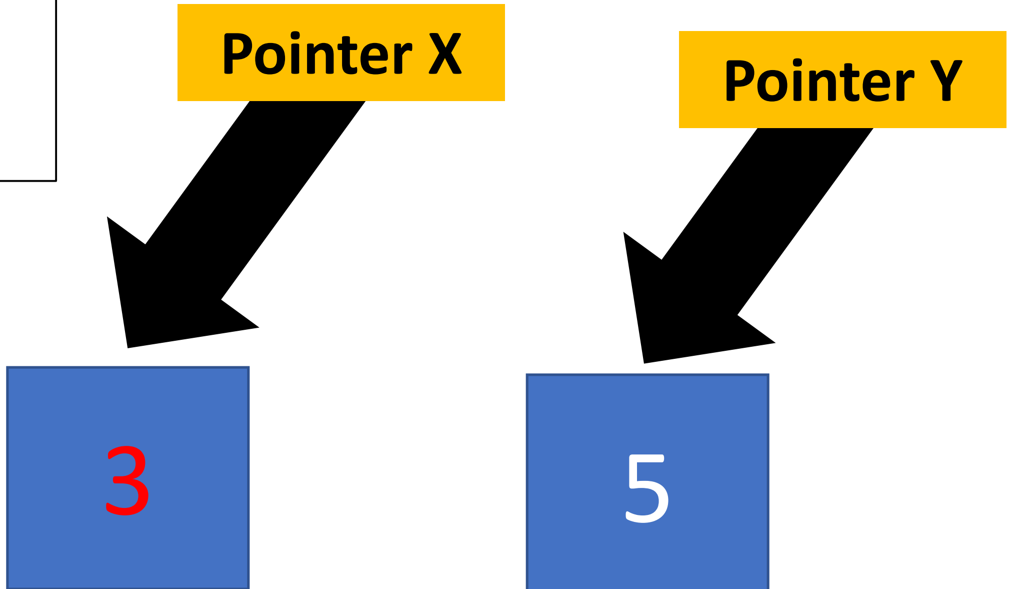


Tracing through the code 3

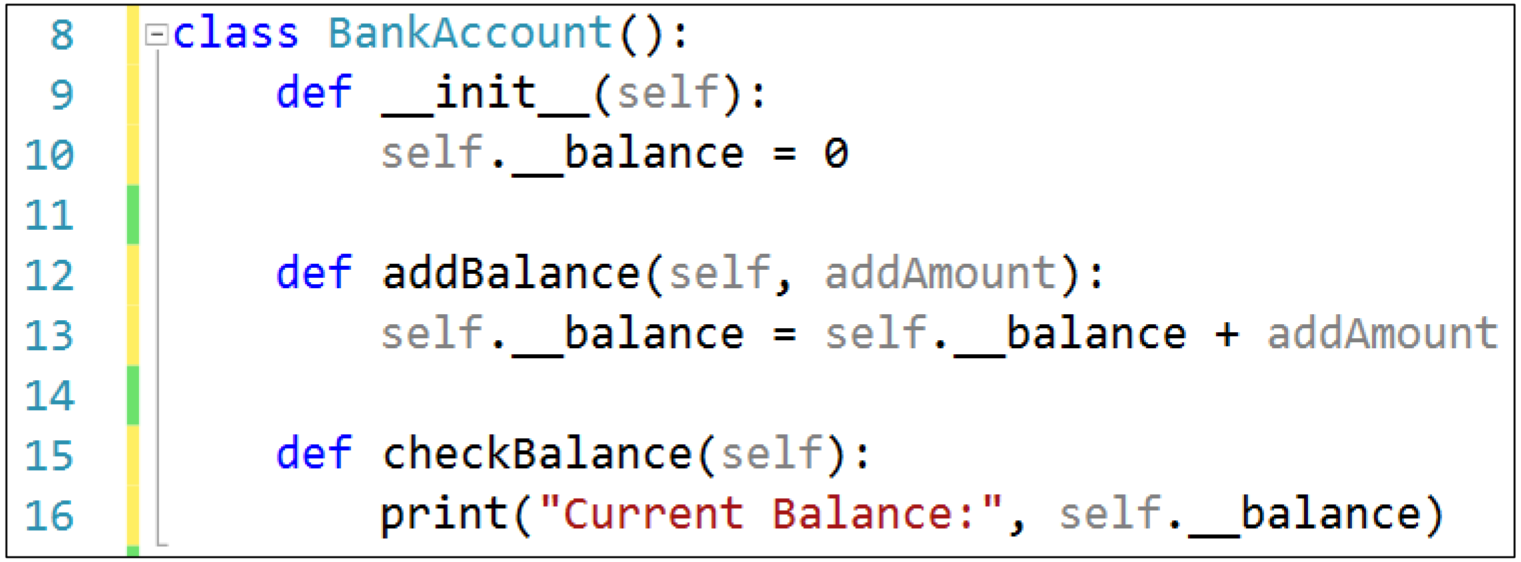


```
x = 5
y = x
x = 3
print(y)
```

- The important thing to note is that “x” is a primitive data type.
- When we use the “=“ sign the <u>*VALUE*</u> of the primitive is copied over to a new cell in memory.



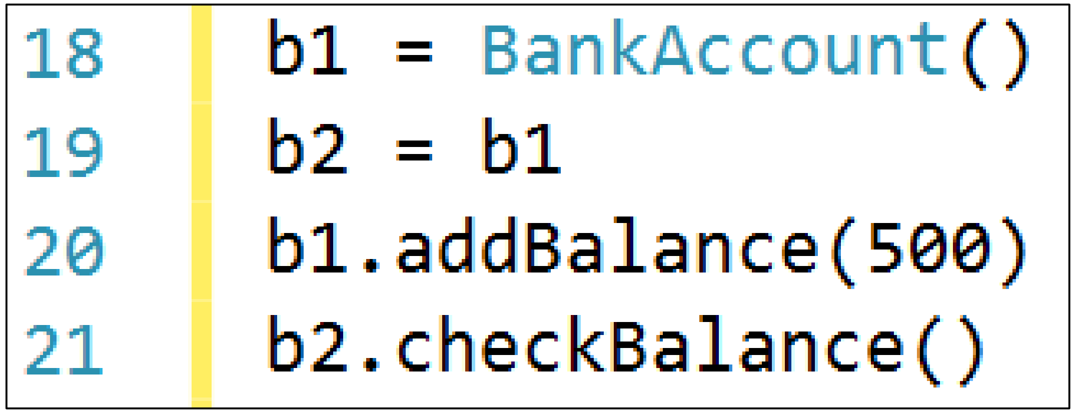
Code Example 2



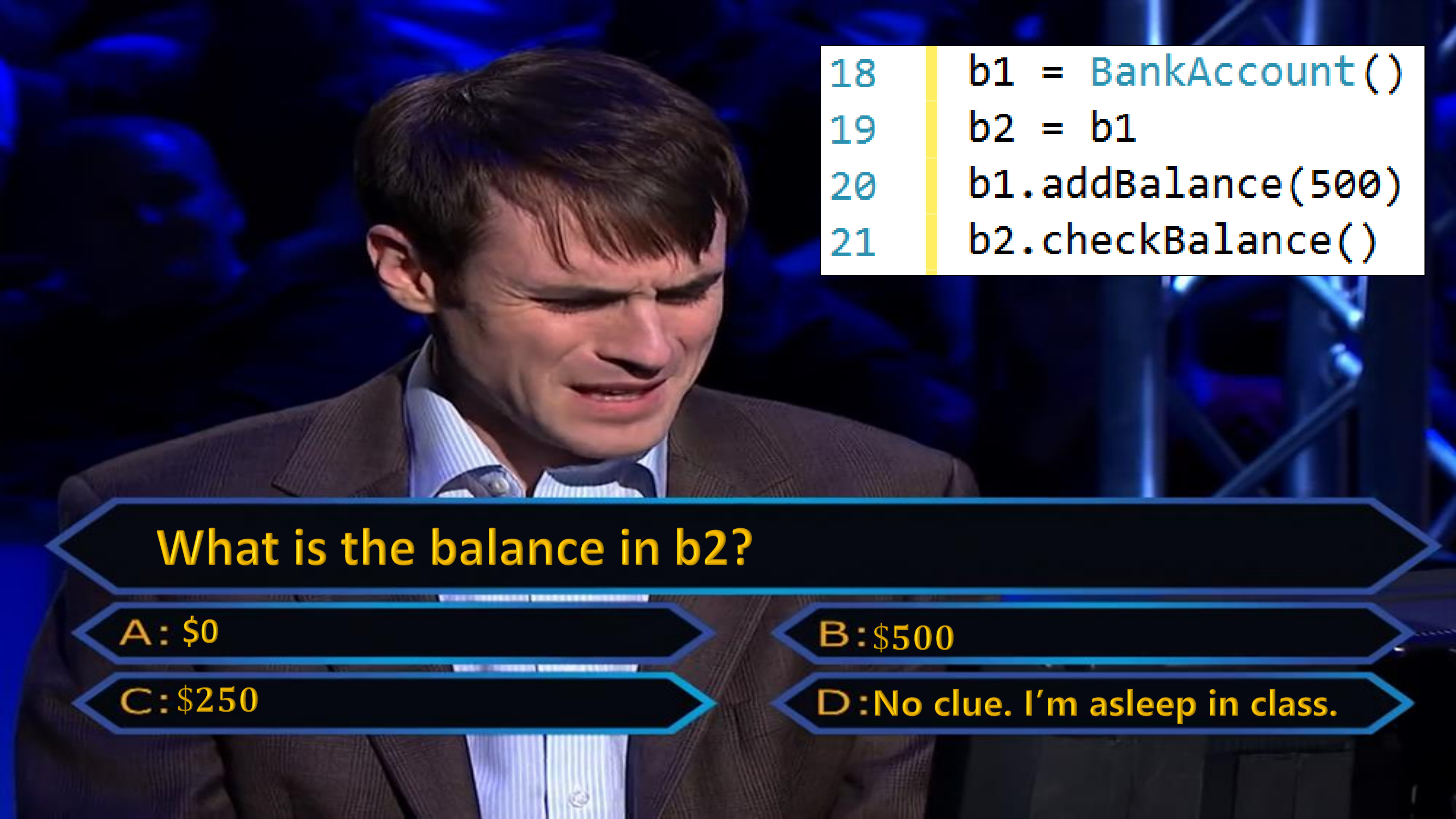
```python
class BankAccount():
    def __init__(self):
        self.__balance = 0

    def addBalance(self, addAmount):
        self.__balance = self.__balance + addAmount

    def checkBalance(self):
        print("Current Balance:", self.__balance)
```



```python
b1 = BankAccount()
b2 = b1
b1.addBalance(500)
b2.checkBalance()
```



```
b1 = BankAccount()
b2 = b1
b1.addBalance(500)
b2.checkBalance()
```

What is the balance in b2?

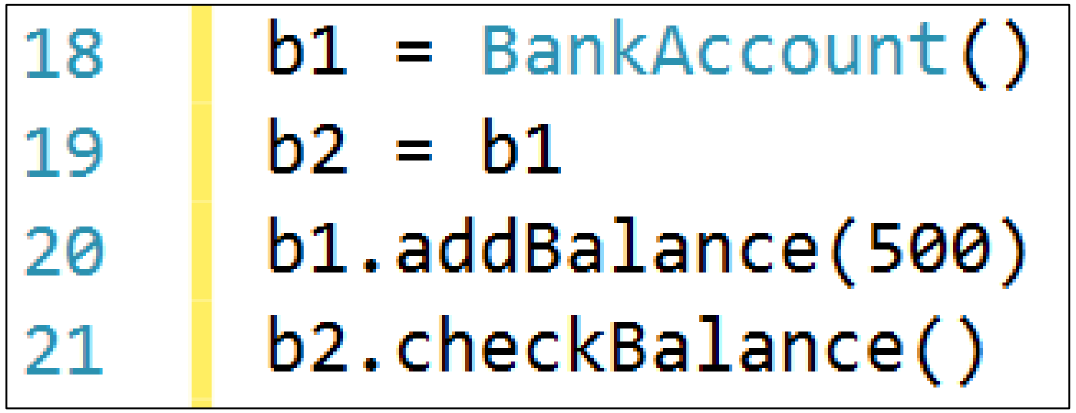
A: $0

B: $500

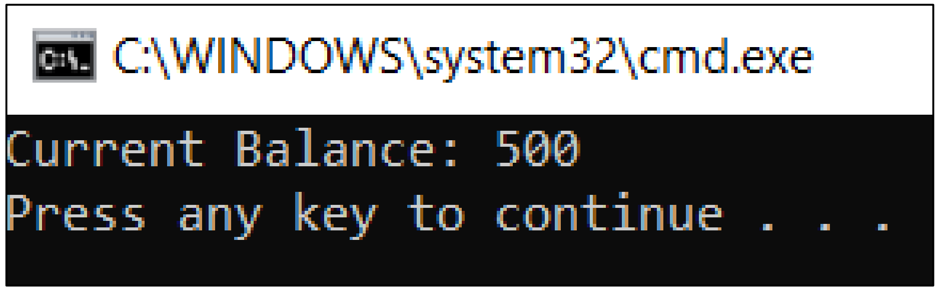
C: $250

D: No clue. I'm asleep in class.

Code Example 2



```
b1 = BankAccount()
b2 = b1
b1.addBalance(500)
b2.checkBalance()
```

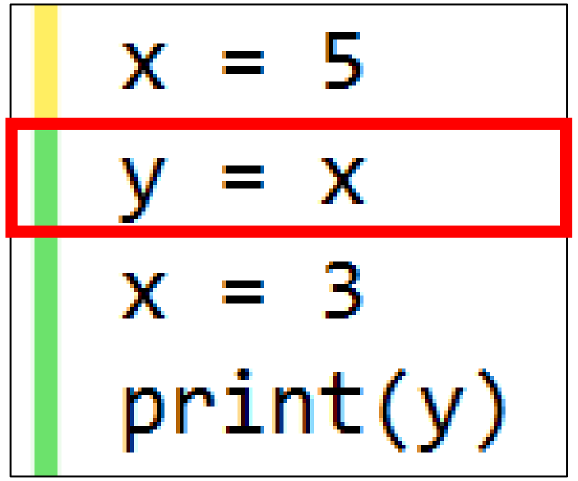


C:\WINDOWS\system32\cmd.exe

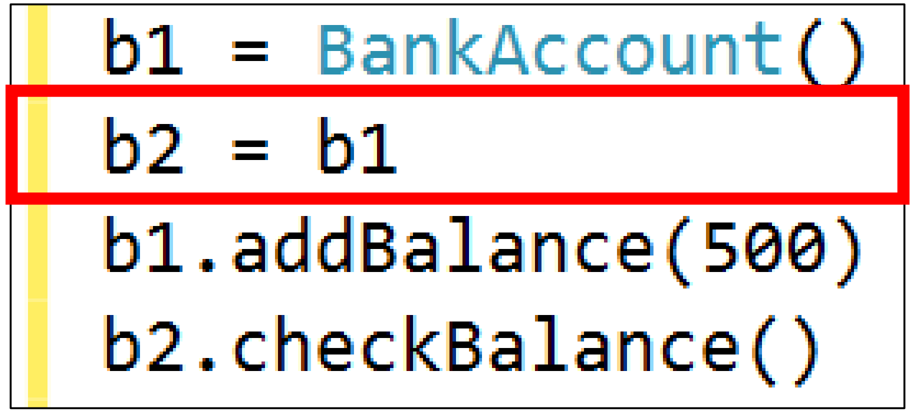
```
Current Balance: 500
Press any key to continue . . .
```

Is anyone *not* confused yet?

What is the difference between these two pieces of code?



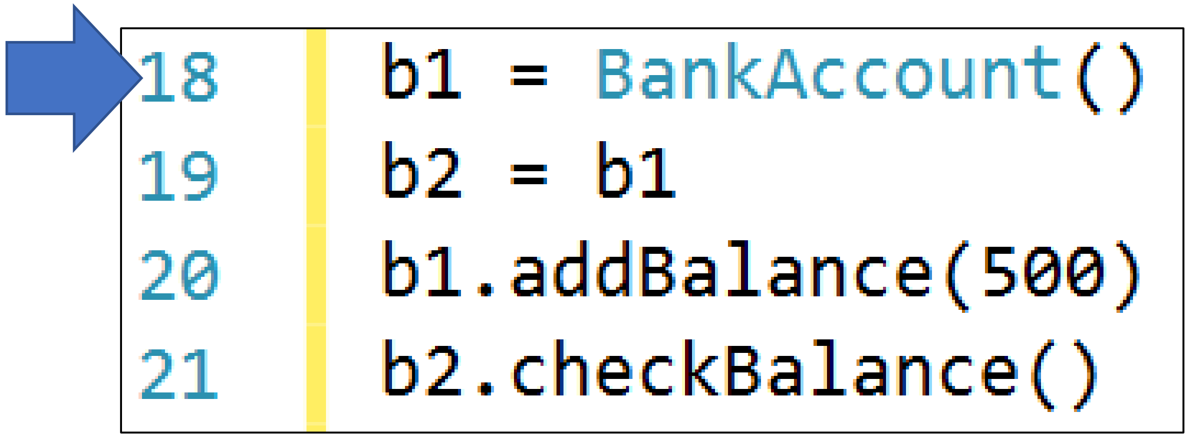
```
x = 5
y = x
x = 3
print(y)
```



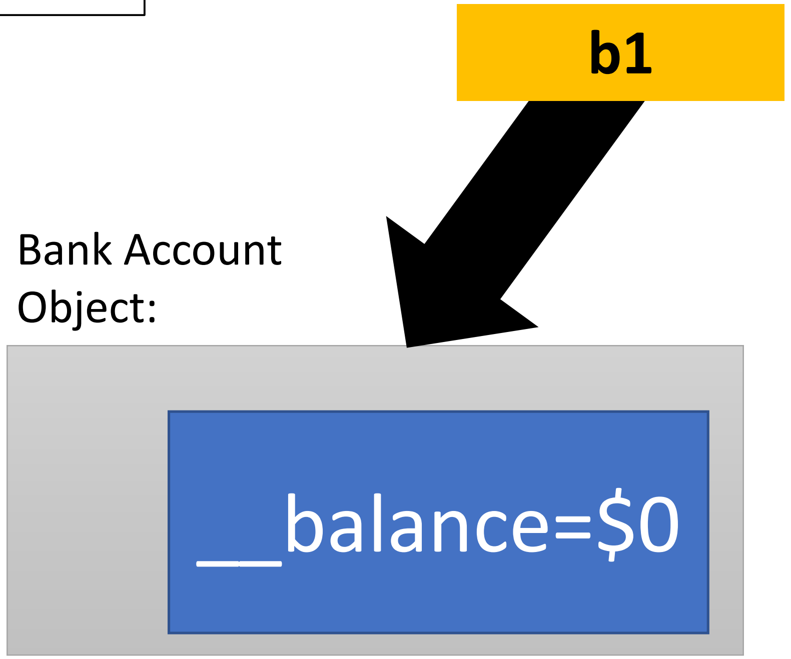
```
b1 = BankAccount()
b2 = b1
b1.addBalance(500)
b2.checkBalance()
```

- Numbers in Python are a primitive datatype. When we use the equality sign it creates a NEW value and a NEW pointer to that value.
- Objects are NOT considered primitive data.
- What happens in when we try to copy objects?

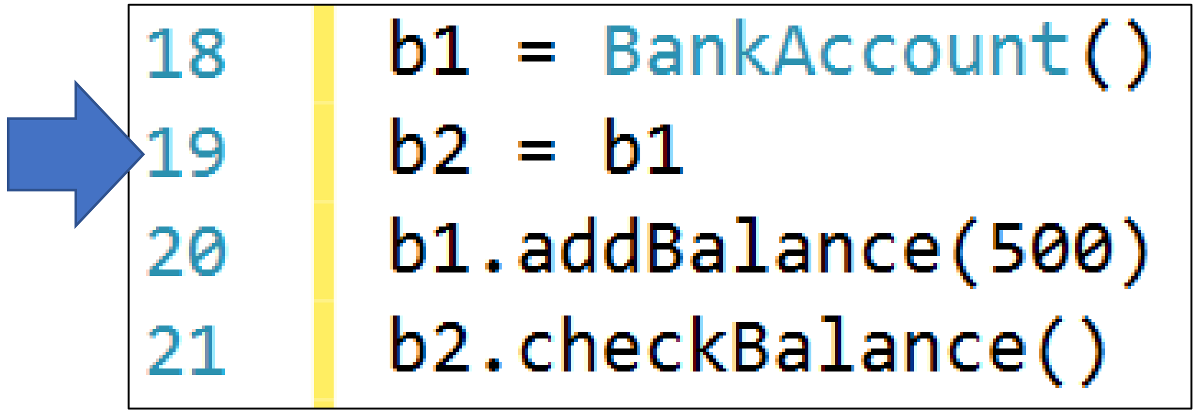
Code Example 2 Revisited



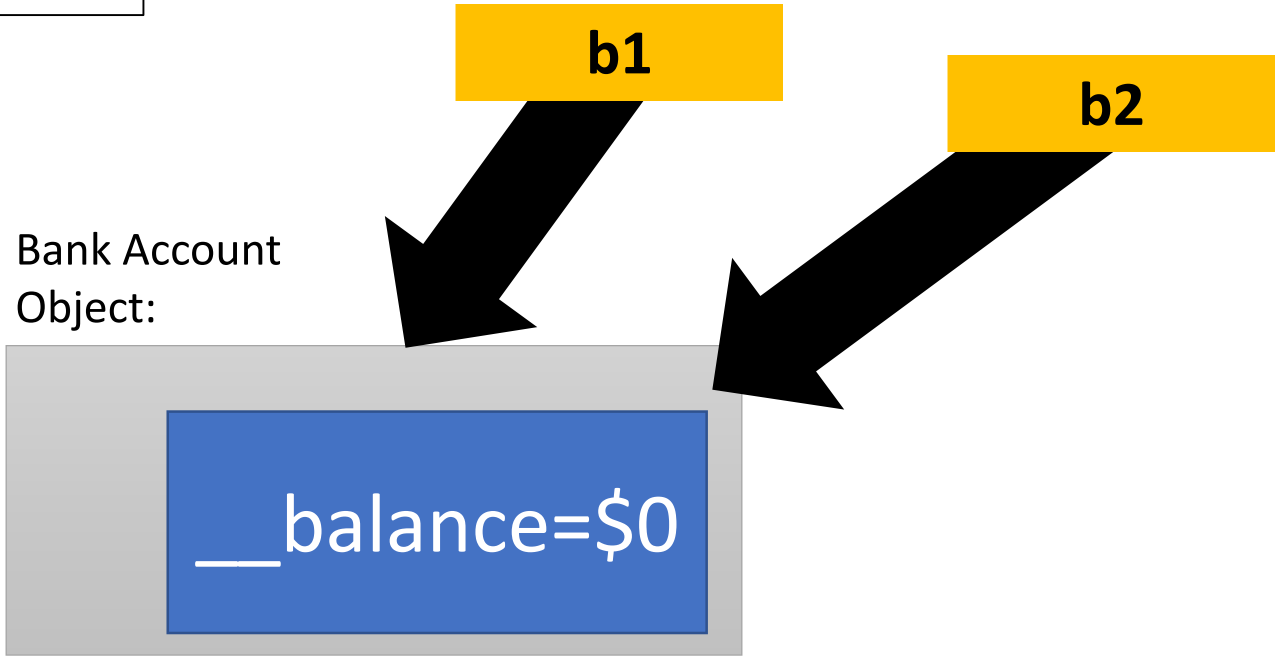
```
b1 = BankAccount()
b2 = b1
b1.addBalance(500)
b2.checkBalance()
```



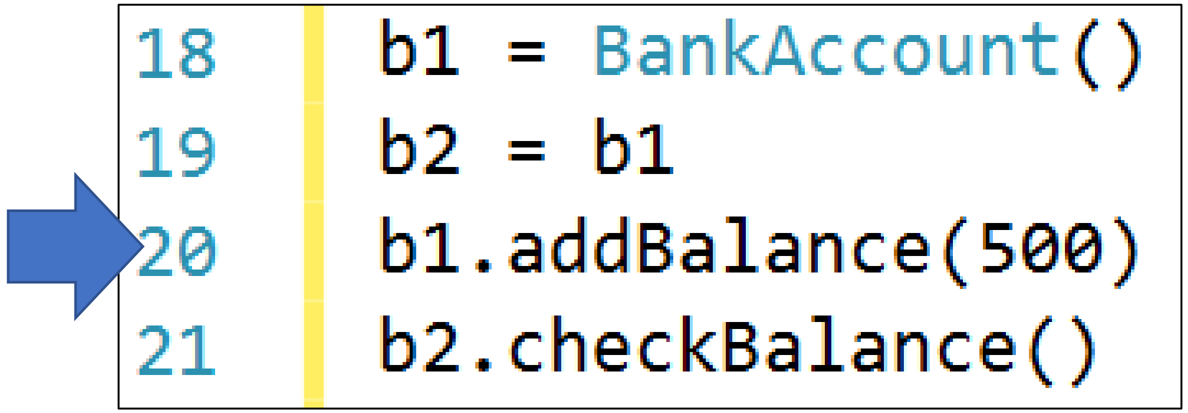
Code Example 2 Revisited



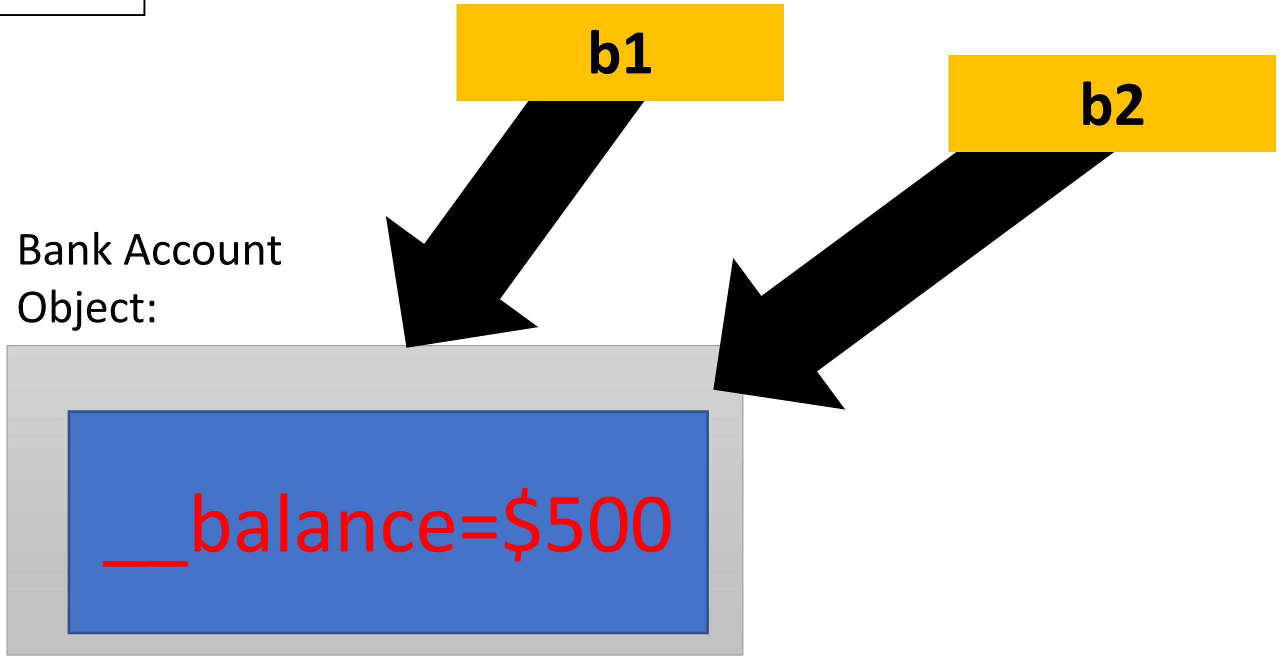
```
b1 = BankAccount()
b2 = b1
b1.addBalance(500)
b2.checkBalance()
```



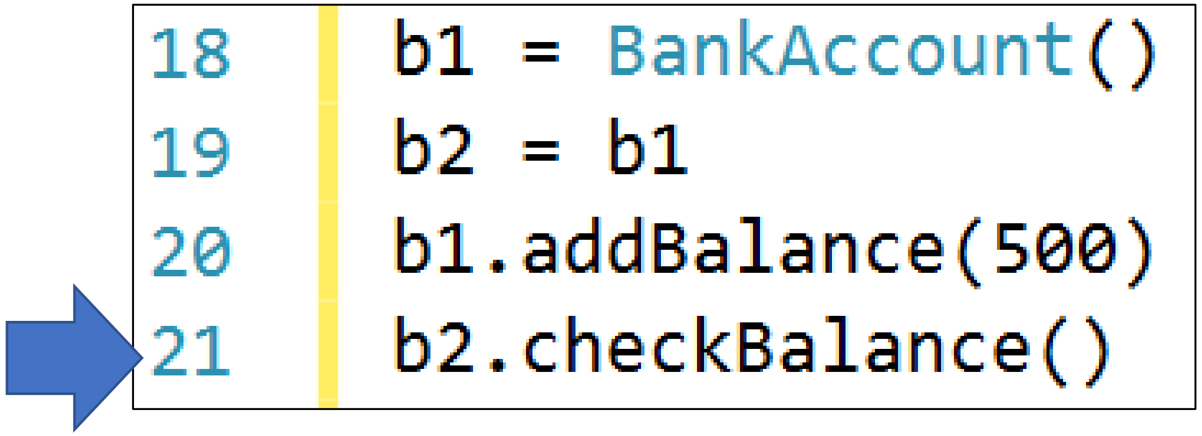
Code Example 2 Revisited



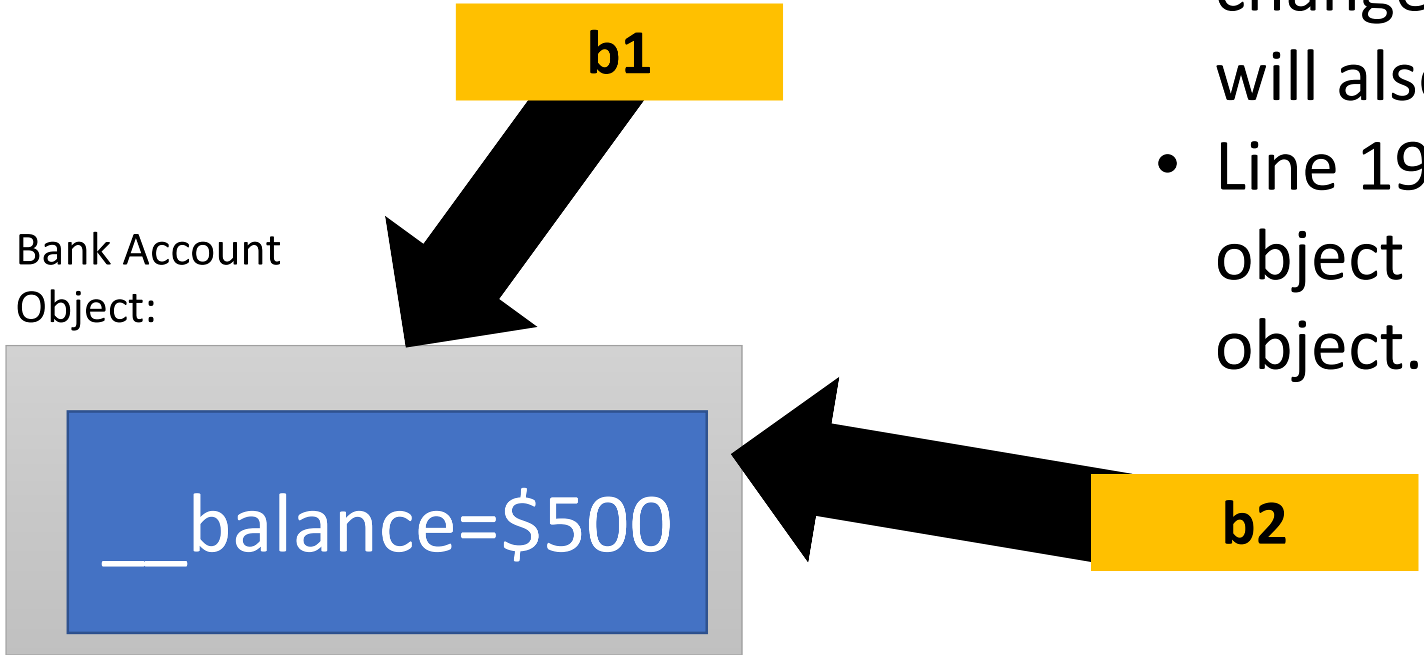
```
b1 = BankAccount()
b2 = b1
b1.addBalance(500)
b2.checkBalance()
```



Code Example 2 Revisited



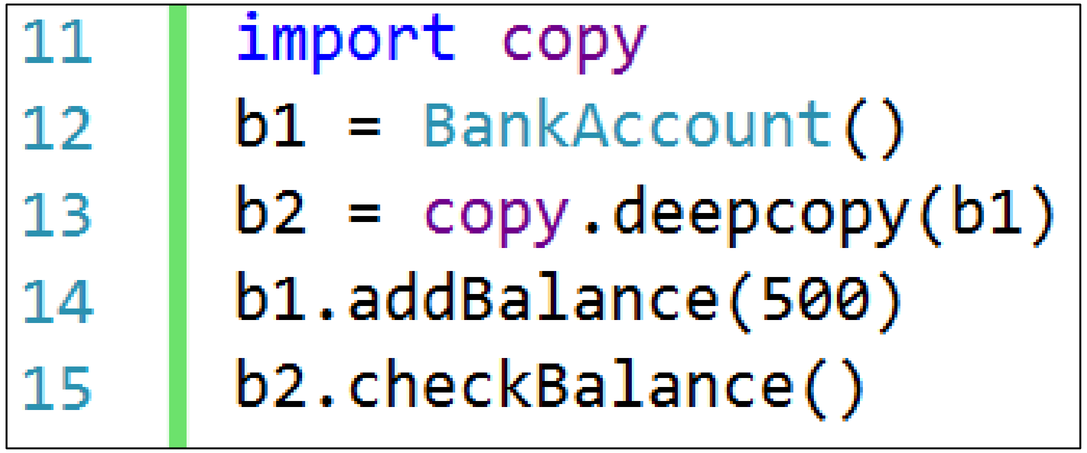
```
18    b1 = BankAccount()
19    b2 = b1
20    b1.addBalance(500)
21    b2.checkBalance()
```



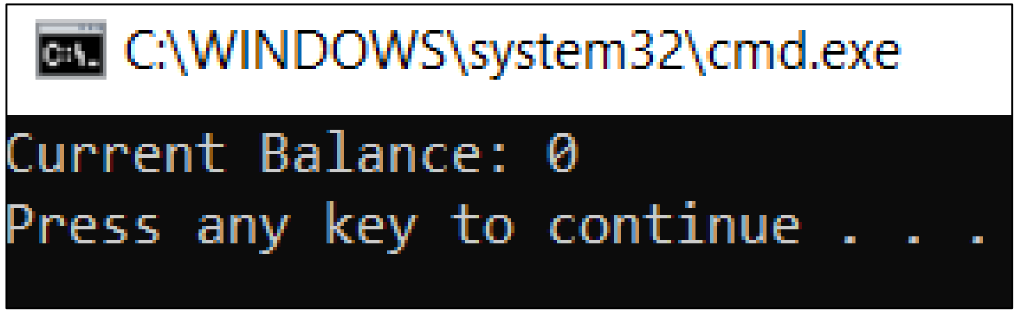
- Object b2 has a balance of $500 because it still points to the SAME space in memory as b1.
- If the value b1 points to gets changed, then when we check b2 it will also have that changed value.
- Line 19 creates a pointer to the object in memory, NOT a new object.

Side note: How can we fix this problem?

The lazy way…
use Python’s built in copy function



```python
import copy
b1 = BankAccount()
b2 = copy.deepcopy(b1)
b1.addBalance(500)
b2.checkBalance()
```

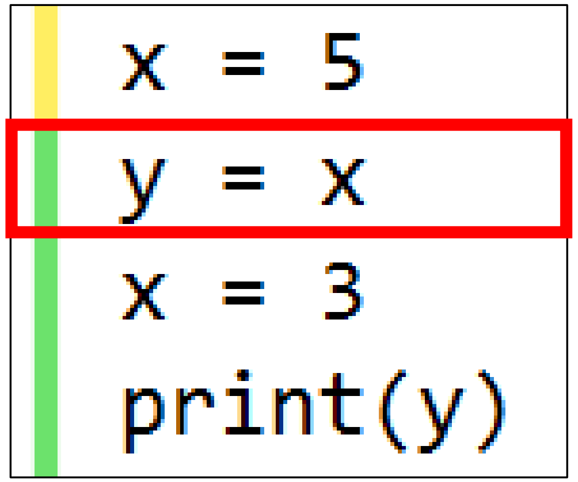


C:\WINDOWS\system32\cmd.exe

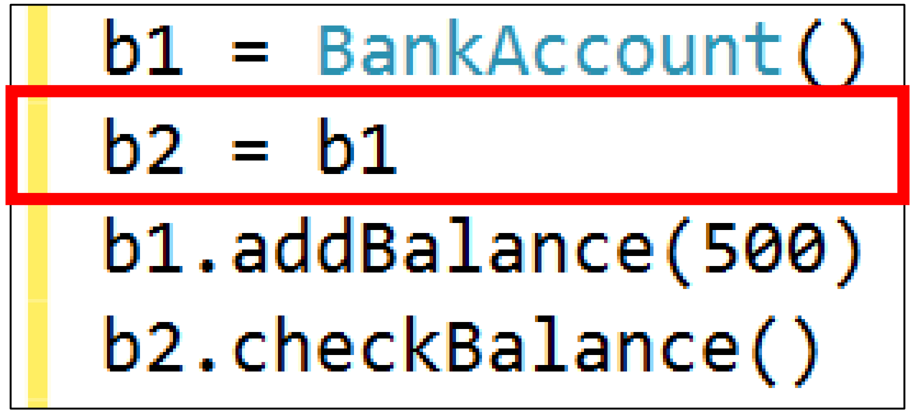
```
Current Balance: 0
Press any key to continue . . .
```

Learn more about copying in Python here: https://docs.python.org/3/library/copy.html

Conclusions from the Primitive and Object Copying Examples



```
x = 5
y = x
x = 3
print(y)
```



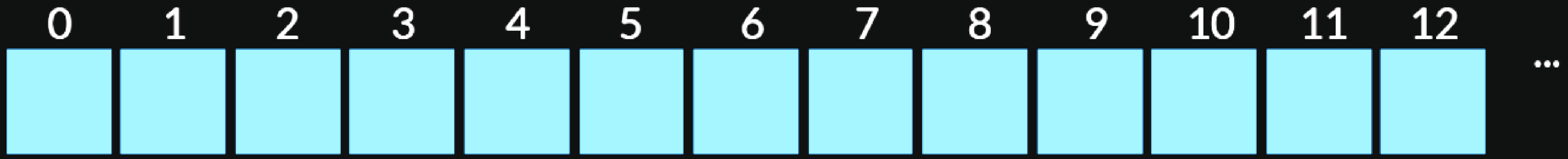
```
b1 = BankAccount()
b2 = b1
b1.addBalance(500)
b2.checkBalance()
```

- Pointers are used to access a certain place in memory.
- When dealing with a primitive, using the equality symbol you get a NEW pointer and a NEW place in memory with the value copied over.
- When dealing with objects, using the equality symbol gets you a NEW pointer to the SAME place in memory.
- This is important!

Now let’s explore the concept of pointers in Scheme…

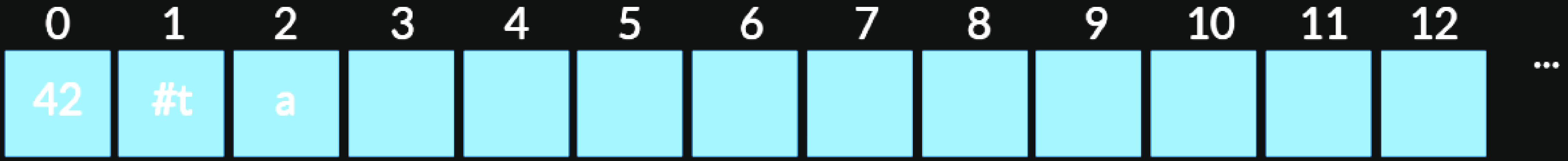
VALUES AND POINTERS IN SCHEME

- You may think of the Scheme interpreter as maintaining an array of memory cells. A memory cell can hold a primitive Scheme value: a Boolean, a character, a token, a number, ... (In fact, it is slightly more complex than this: some primitive values may require several adjacent cells.)
- Each memory cell has a unique address--a number that the computer uses to refer to it.

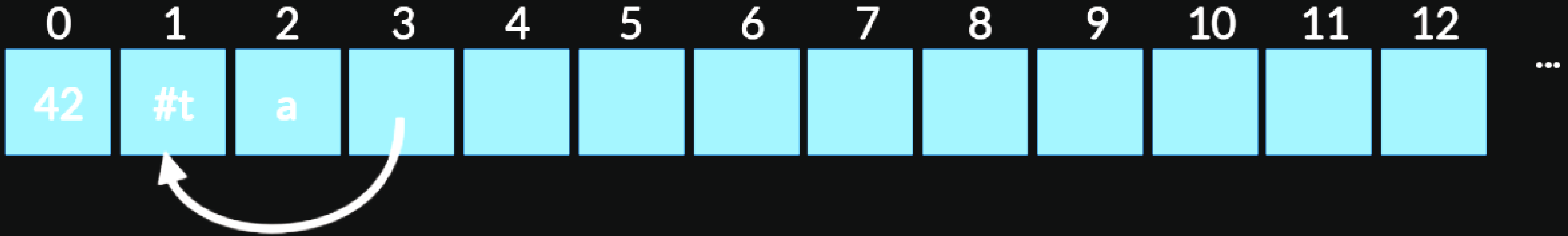


CELLS HOLD VALUES OR POINTERS

- A cell can hold an primitive value.

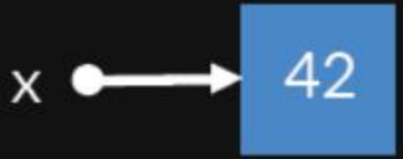


- A Scheme variable is associated with a cell that holds *a pointer containing the name of another cell that holds its value.* [redacted]

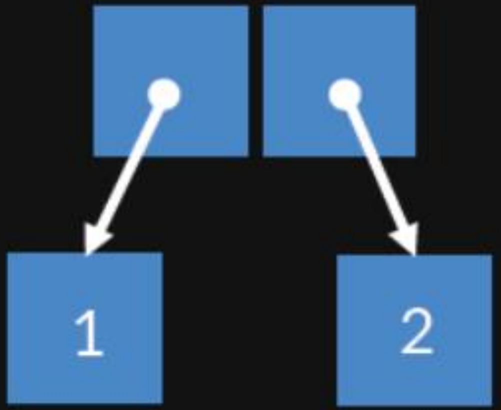


REPRESENTING VARIABLES AND PAIRS

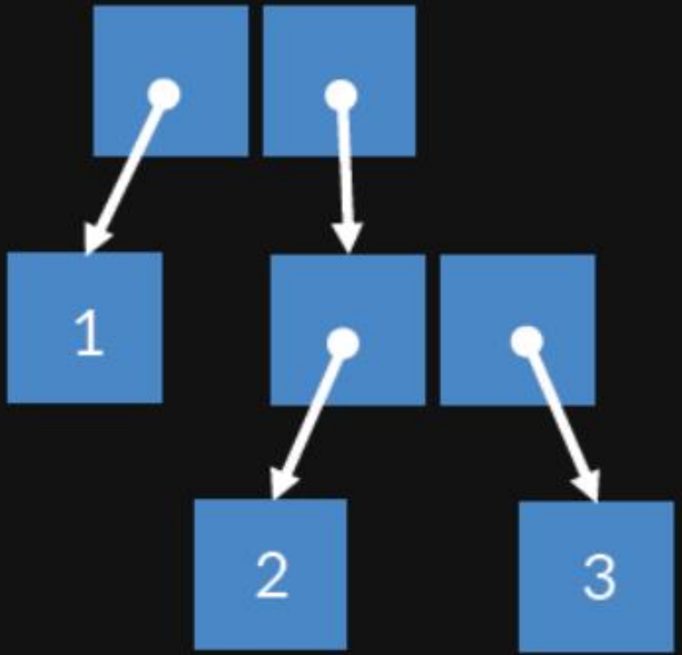
- A mnemonic way to represent a variable in Scheme, then:



- This same convention is used for pairs. A pair in Scheme is represented as a tuple of *two pointers*, one pointing to the contents of the car, one to the cdr. Thus you may think of a pair as occupying two adjacent memory locations, each a pointer. Thus...



```
(1 . 2)
```



```
(1 . (2 . 3))
```

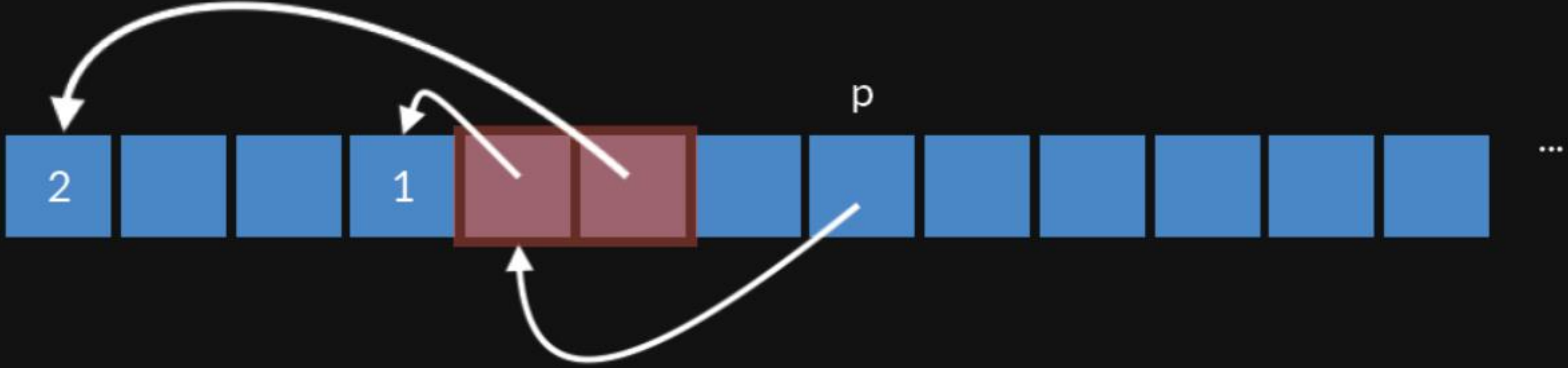
PAIRS AS *PAIRS OF POINTERS*

- Consider the code snippet

```
(define p (cons 1 2))
```

- This definition yields a memory layout something like this:

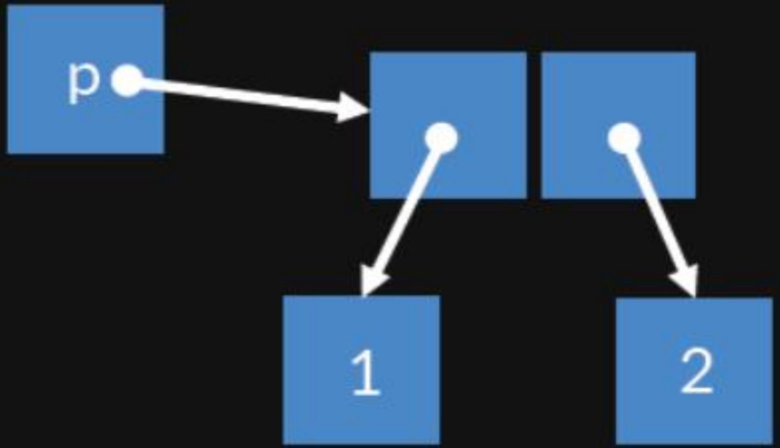
```
(1 . 2)
```



However, we typically abstract away the exact positions, and simply draw relevant memory cells...

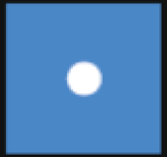
- Consider the code snippet `(define p (cons 1 2))`

- This definition yields a memory layout something like this: `(1 . 2)`



RECALL THE SCHEME CONVENTION FOR LISTS

- Scheme reserves a special null pointer.

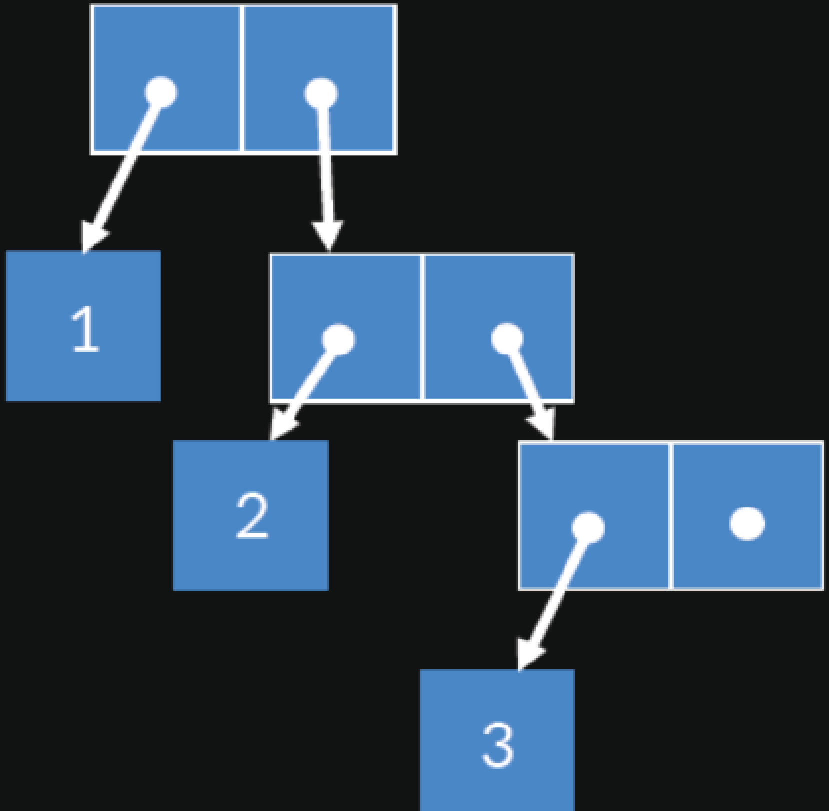


- Then, lists are represented as nested pairs, where the car points to the first element of the list and the cdr points to the "rest" of the list.

```
(1 . (2 . (3 . '())))
```

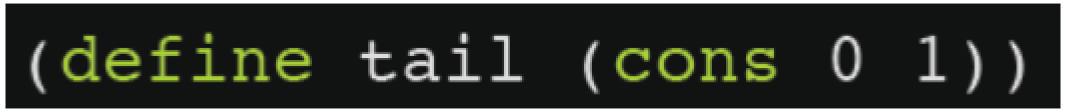
or

```
(1 2 3)
```

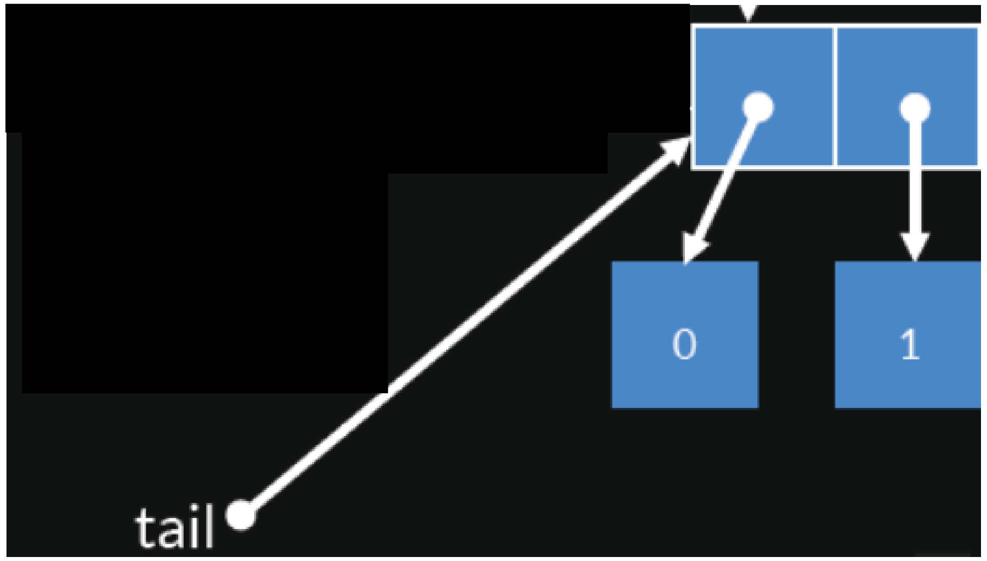


Using the concept of pointers we can create variables which have certain links

Right now it will not be apparent WHY we might need to link variables but it may become clear later when we return to the queue example.



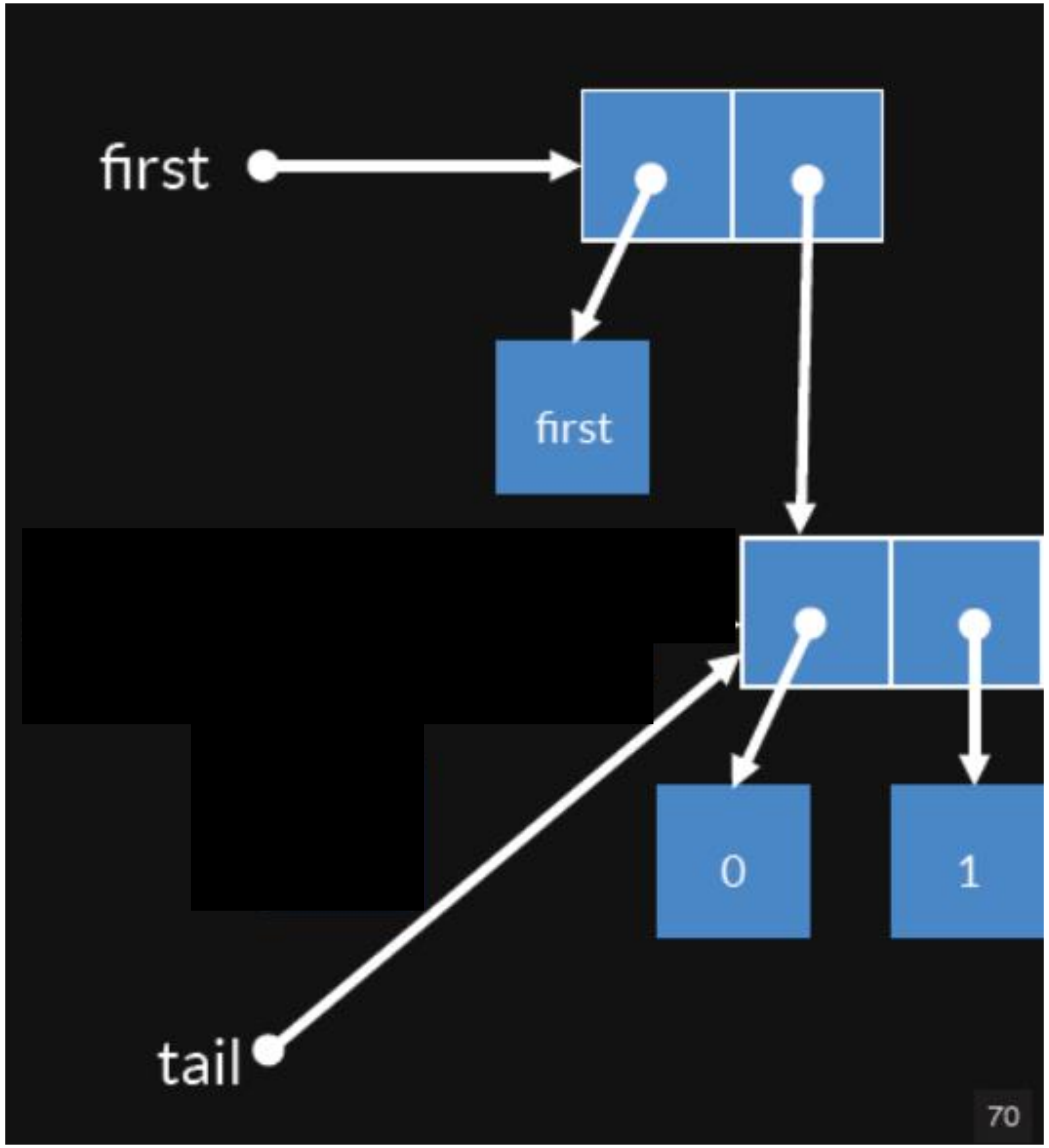
```
(define tail (cons 0 1))
```



```
(define tail (cons 0 1))
```

```
(define first (cons 'first tail))
```

We now have linked first and tail together…



[redacted]

```
(define tail (cons 0 1))
```

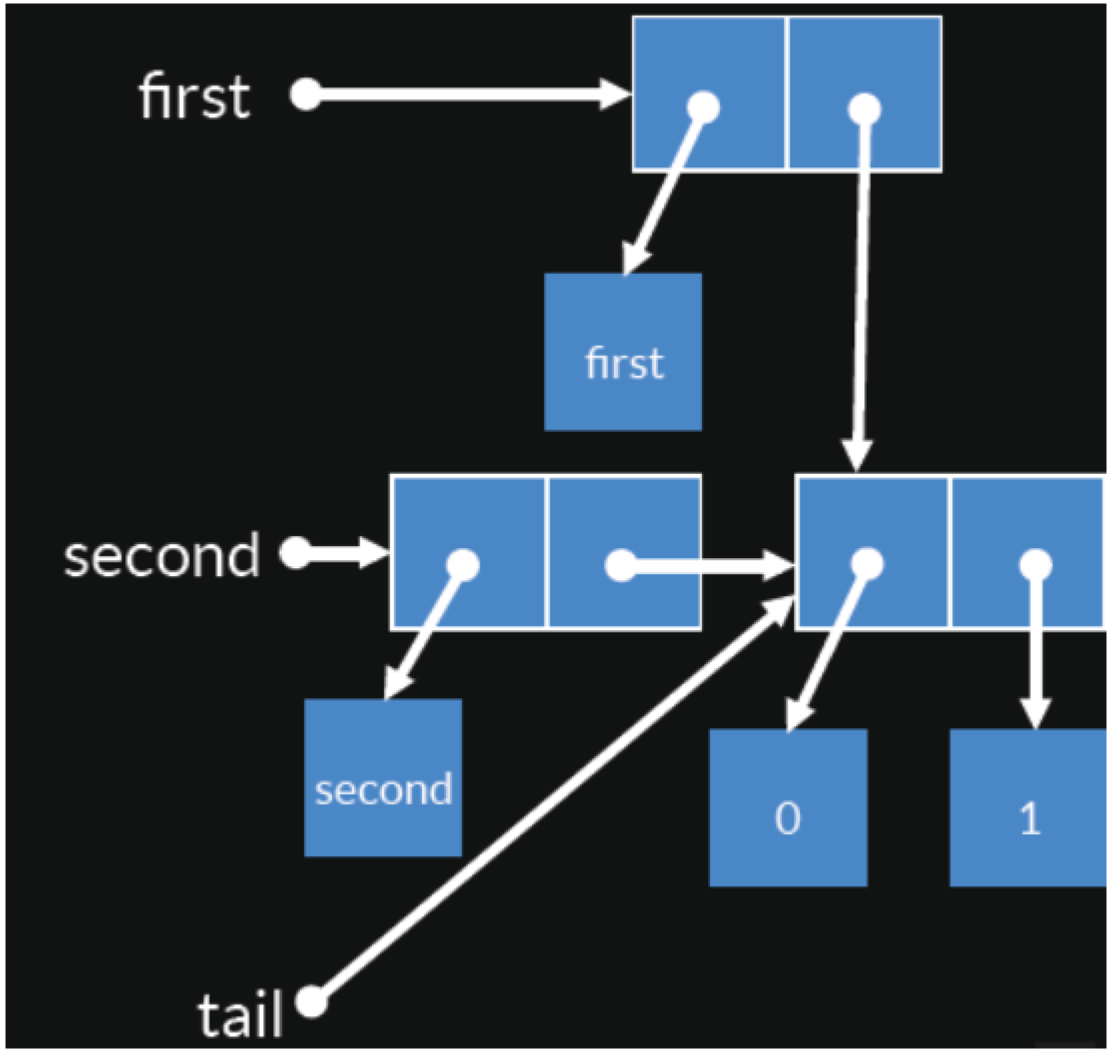
```
(define first (cons 'first tail))
```

We now have linked first and tail together.

```
(define second (cons 'second tail))
```

We now have linked second and tail together.

```
> first
(first 0 . 1)
> second
(second 0 . 1)
```



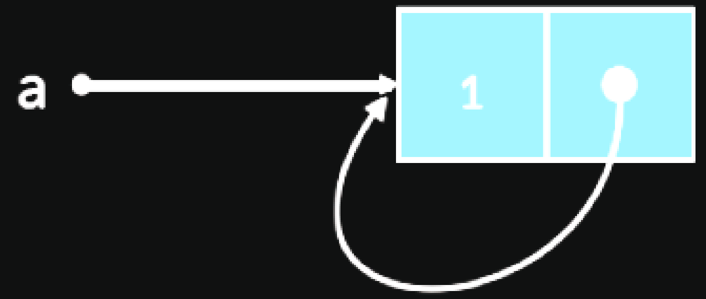
Preview of the motivation for doing this: Let’s say you were adding elements to an array and wanted to keep track of the FIRST and LAST element. However you only added in one element. Wouldn’t the tail and first all pointing to the same thing?

Set! / cdr / car

- You’ve seen set! in action (on a variable). It redirects the pointer so that it points to a new object.
- There are analogous operations for destructively setting the car and cdr of a pair: set-car! and setcdr!

THIS CAN CREATE (ARBITRARILY) COMPLEX, EVEN CIRCULAR, STRUCTURES

```
> (define a (list 1))
> (set-cdr! a a)
> (car a)
1
> (cadr a)
1
> (caddr a)
1
> (caddr a)
1
> (cadddr a)
1
```



This creates a kind of infinite data structure.

A preview of where we are going…

	Array		Queue
Insertion	O(n)	Insertion (Enqueue)	O(1)
Deletion	O(n)	Deletion (Dequeue)	O(1)

Figure Sources

- https://www.varsity.co.uk/images/ecms/2021/03/icons8-team-r-enAOPw8Rs-unsplash-scaled.jpg
- https://imgflip.com/s/meme/Left-Exit-12-Off-Ramp.jpg
- https://preview.redd.it/8cyj4thfoqd31.png?auto=webp&s=21b1c8530cde550bcc3e4567c7763d486184e325