ICT162

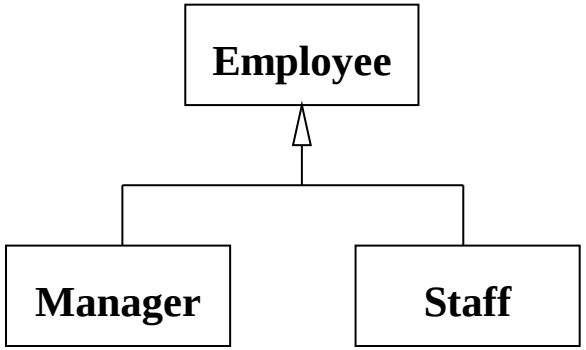
Object Oriented Programming

Seminar 2 Composition and Collection

Introducing Reuse by Composition

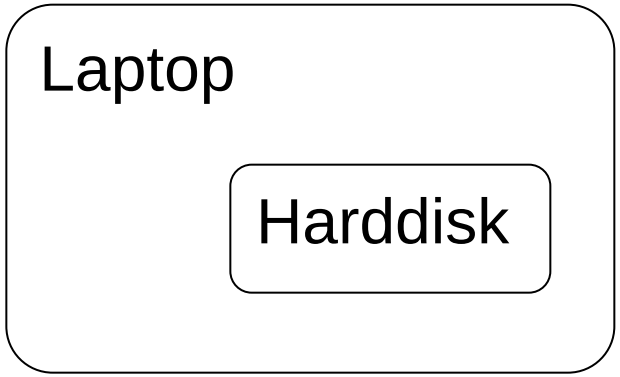
Re-use by Inheritance

"is a" relationship



Re-use by Composition

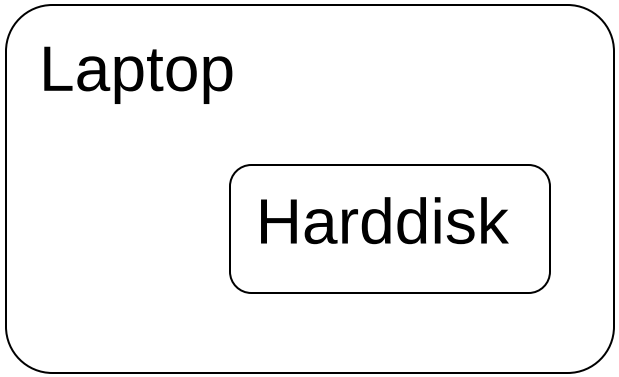
"has a" relationship



Software Reuse

Re-use by Composition

"**has a**" relationship



Different in nature

Hidden inside

Make use of function

save in laptop → save in harddisk

Object Composition

- To model a customer who owns a cash card
- The customer “has-a” cash card

```
class Customer:
    def __init__(self, name, card):
        self._name = name
        self._card = card
```

Creating a Customer Object

- Create the CashCard object first

  c = CashCard(*'1', 10.0*)

- Create the Customer object

  cust = Customer(*'James', c*)

Another way to write the Constructor

```
class Customer:
    def __init__(self, name, id , amt):
        self._name = name
        self._card = Card(id, amt)
```

- To create the Customer object

  cust = Customer('Jack', '2', 50)

- Which constructor to specify depends on the application

Using Default Parameters

```
class Customer:
    def __init__(self, name, card = None, id = None, amt = None):
        self._name = name
        if card is None:
            self._card = CashCard(id, amt)
        else:
            self._card = card
```

```
c = CashCard('1', 10.0)
cust = Customer('James', c)
cust = Customer('Jack', id = '2', amt = 50)
```

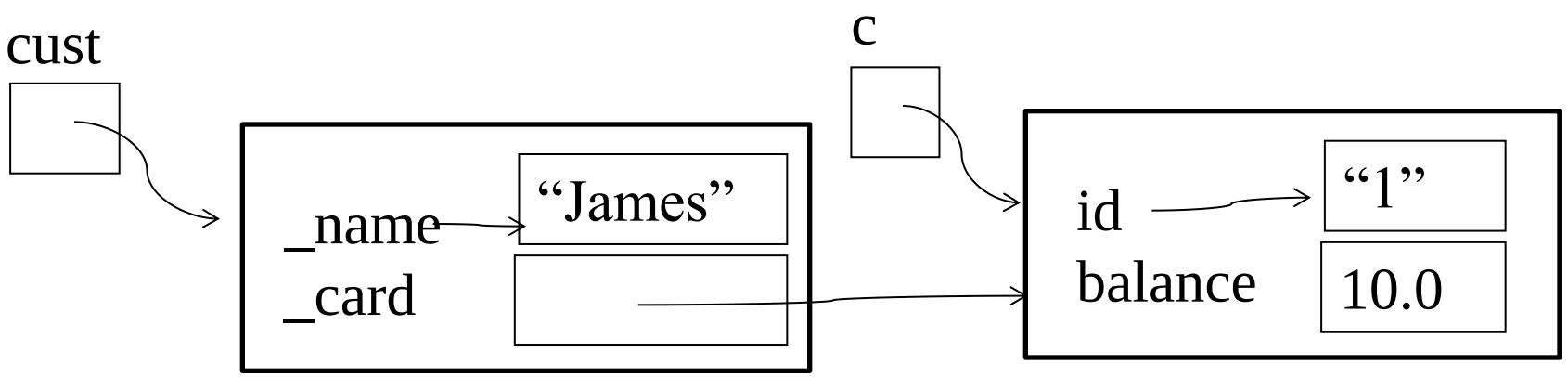
Getter and setter methods for CashCard

```
@property
def card(self):
        return self._card

@card.setter
 def card(self):
        return self._card
```

Other methods

```
def topUp(self, amt):
    self._card.topUp(amt)

def cardBalance(self):
    return self._card.balance
```

Object Composition



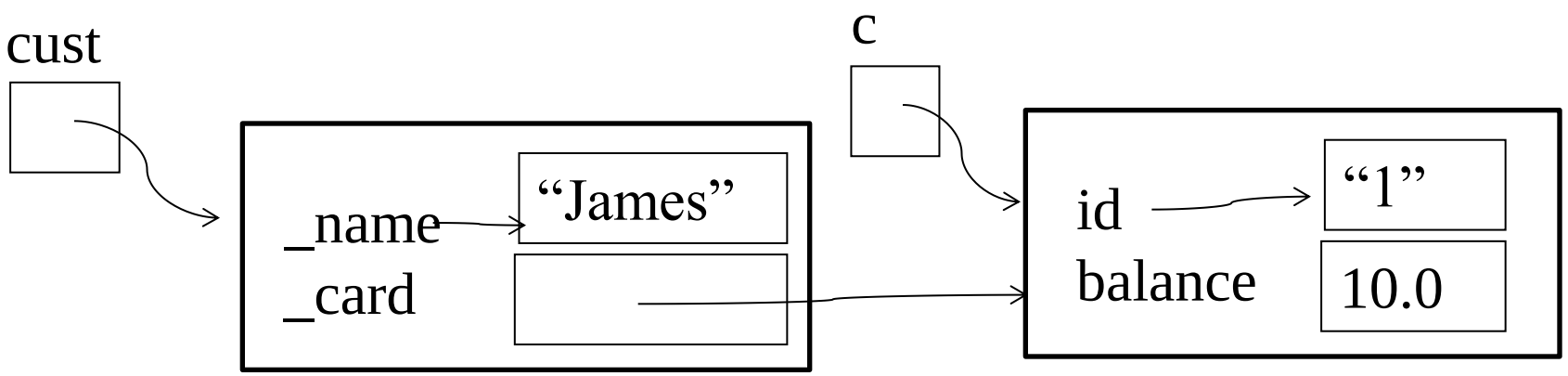

```
def cardBalance(self):
    return self._card.balance
```

```
@property
def balance(self):
    return self.__balance
```

```
c = CashCard('1', 10.0)
cust = Customer('James',c)
print(cust.cardBalance())
```

→ `self._card.balance`

Object Composition




```
def topUp(self, amt):
    self._card.topUp(amt)
```

```
def topUp(self, amount):
    self._balance += amount
```

```
c = CashCard('1', 10.0)
cust = Customer('James',c)
cust.topUp(20)
```

→ `self._card.topUp(20)`

Collection of Objects

- List or dictionary can be used to keep track of objects
- E.g.

```
cards = []
```

- To add an object, use append or insert

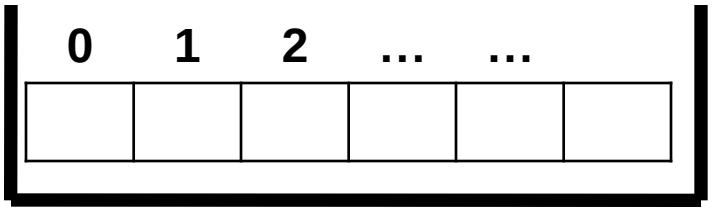
```
cards.append(CashCard(…))
```

Collection of Objects

```
cards = []
cards.append(CashCard(“11”, 10))
cards.append(CashCard(“12”, 20))
cards.append(CashCard(“13”, 30))
…

for c in cards:
    print(c.balance)
```

Creating a Collection

- cards = []



- Objects are added from index 0.

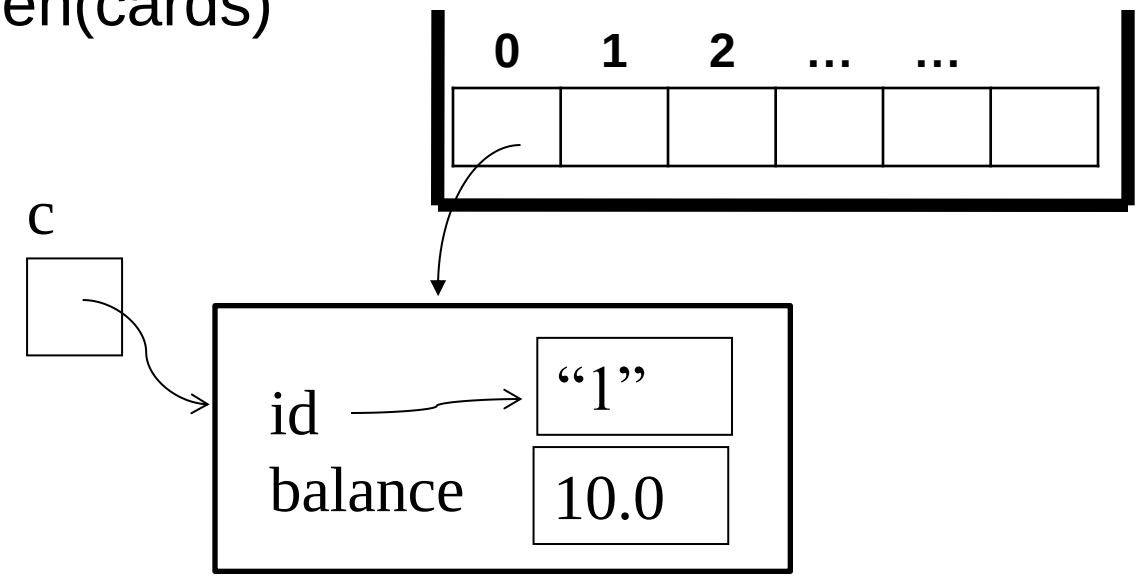
Adding an object to a Collection

- To add a card to the list

  cards.append(c)

- To determine the number of objects in the list,

  num = len(cards)



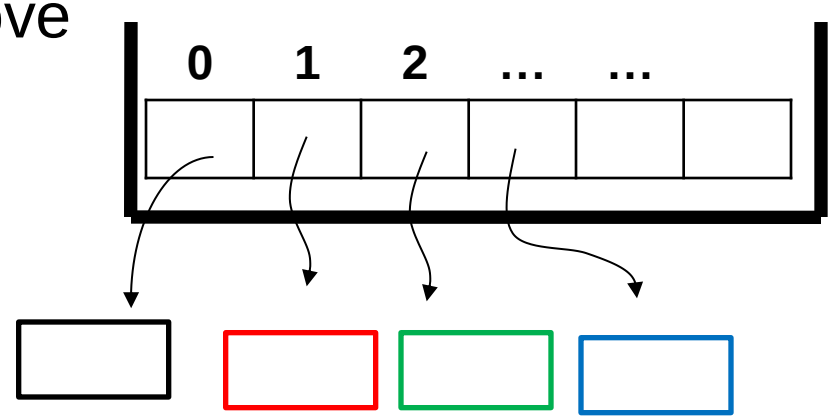
To remove objects from a Collection

- remove(value)
  - remove the first matching *value*, not a specific index. E.g, cards.remove(c)
- del aList[index]
  - remove the value at a specific index. E.g., del cards[0]
- aList.pop(index)
  - remove the item at a specific index and returns it. E.g., cards.pop(0)

Whenever an object is removed from an index, objects on the right are shifted one left to that index

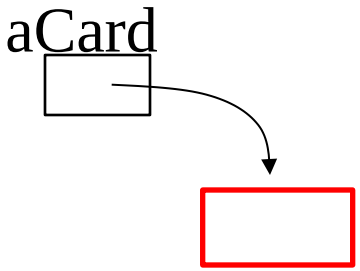
To remove objects from a Collection

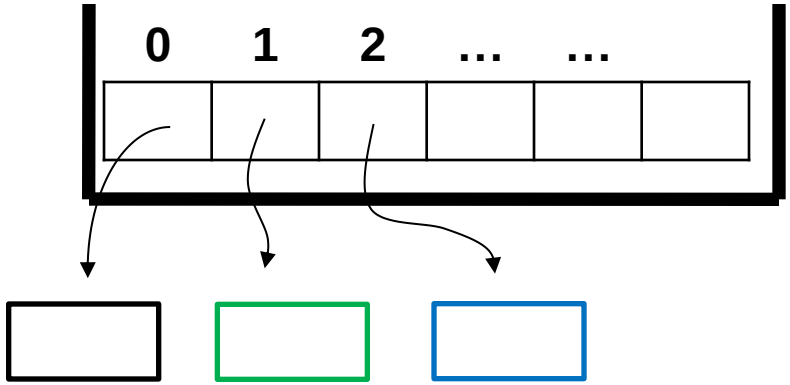
- Before remove



- After remove

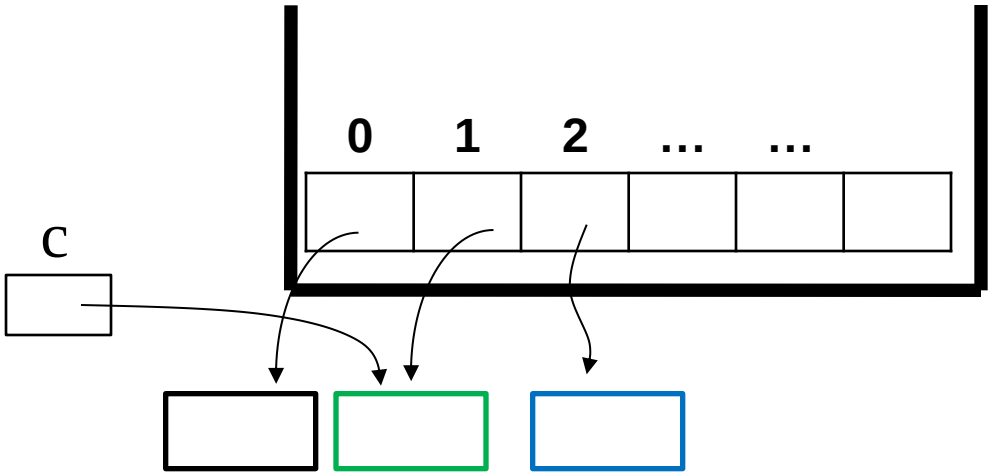
```
aCard = cards.pop(1);
```





To retrieve object from a Collection

- To retrieve one object, specify the index
  c = cards[1]
- Object not removed! Only the reference assigned to c.



Iterate a Collection

- Use a loop to retrieve one object at a time

```
for i in range(len(cards)):
    print(cards[i])
```

- Alternatively,

```
for c in cards:
    print(c)
```

Searching and removing from a Collection

- To determine if an object is in the list,

  c in cards

- To determine if an object is in the list, then remove it

```
if c in cards:
    cards.remove(c)
```

To search object in a Collection given its identifier

```
found = False
for c in cards:
        if c.id == id:
            found = True
            break
if found:
    print(c)
else:
    print('Not found!')
```

Alternatively,

```
for c in cards:
    if c.id == id:
        print(c)
        break
else:
    print('Not found!')
```

Dictionary Collection of Objects

```
cards = {}
c1 = CashCard(“11”, 10)
c2 = CashCard(“12”, 20)
c3 = CashCard(“13”, 30)
…
cards[c1.id] = c1   #Adding
cards[c2.id] = c2
cards[c3.id] = c3

for c in cards.values():  #Iterate
    print(c.getBalance())
```

Dictionary operations

```
uid = input('Enter card id: ') #locating
print(cards.get(uid, 'Not found')

If uid in cards: #check membership
   print('in collection')

if uid in cards:  #removing
   del cards[uid]
```

Object Composition using a List – PhoneBook application

- An application to keep track of Contacts in a Phone Book using a list

```
class Contact:
    def __init__(self, name,  phone):
        self._name = name
        self._phone = phone

    @property
    def name(self):
        return self._name

    @property
    def phone(self):
        return self._phone

    @phone.setter
    def phone(self, phone):
        self._phone = phone

    def __str__(self):
        return 'Name: {} Phone: {}'.format(self._name, self._phone)
```

PhoneBook class

A Phone Book class uses a list to store contacts

```
class PhoneBook:
    def __init__(self, name):
        self._name = name
        self._contacts = []
```

Add a Contact to Collection

```
def addContact(self, contact):
    self._contacts.append(contact)
```

Search Contact in Collection

```
def searchContact(self, name):
    for c in self._contacts:
        if c.name == name:
            return c
    return None
```

Remove a Contact in Collection

```
def removeContact(self, name):
        c = self.searchContact(name)
        if c is None:
            return False
        self._contacts.remove(c)
        return True
```

Update a Contact in Collection

```
def updateContact(self, name, phone):
        c = self.searchContact(name)
        if c is None:
            return False
        c.phone = phone
        return True
```

__str__ Method

```
def __str__(self):
        contacts = [str(c) for c in self._contacts]
        contacts.sort()
        return 'Phone Book Owner Name: {}\n{}'
.format(self._name, '\n'.join(contacts))
```

Testing the PhoneBook class

```
def main():
    myFriends = PhoneBook('Jim')

    myFriends.addContact(Contact('Peter', 9123123))
    myFriends.addContact(Contact('Joe', 8123123))
    myFriends.addContact(Contact('Amy', 6123231))
```

Testing the PhoneBook class

```
c = myFriends.searchContact('Joe')
if c is None:
    print('Not found')
else:
    print('number is', c.phone)

myFriends.removeContact('John')
myFriends.updateContact('Amy', 6123456)
print(myFriends)
```