Contact

Dr. James Shackleford
shack@drexel.edu
Bossone 211

Office Hours: 3 – 5 pm (Wednesday)
Course Website: http://learn.dcollege.net

Textbook (for this review)

Think Python
by Allen Downey
O'Reilly Press, 2015
ISBN-13: 978-1449330729
(Freely available in PDF format, check course website)

Grading

(subject to change)

- 30% In-lab Programming Assignments
- 30% Take-Home Programming Assignments
- 40% Programming Projects

Python Uses Duck Typing

Duck Typing

If an object has the method(s) you are looking for

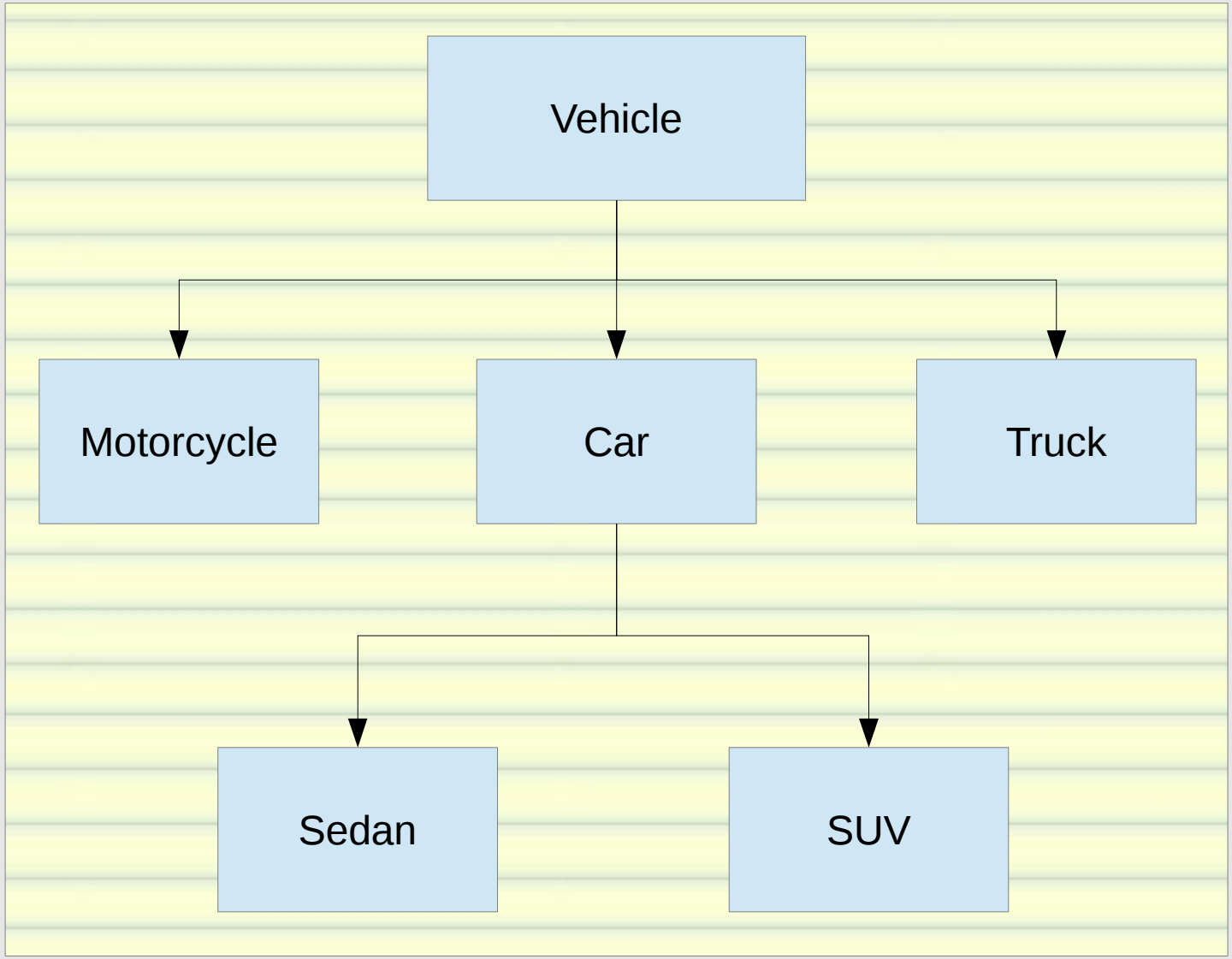
...then it's the right type!

Polymorphism

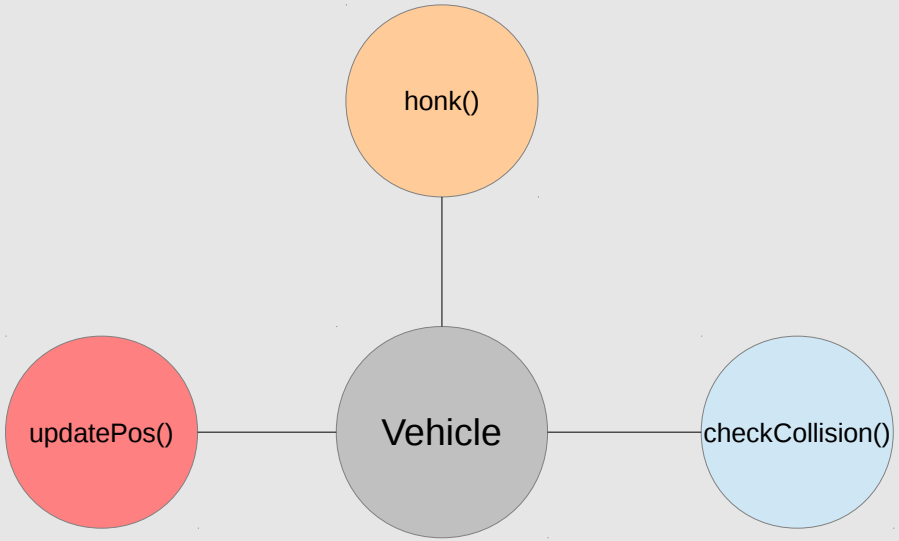
Different Objects (usually inherited).

Common Methods Interfaces.

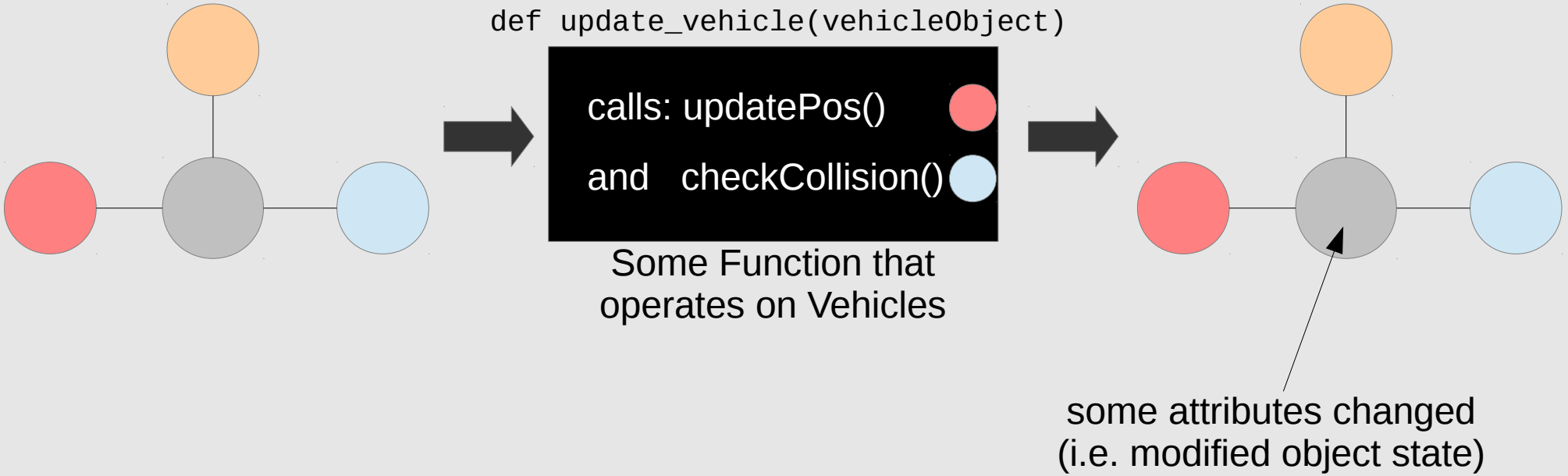
Polymorphism



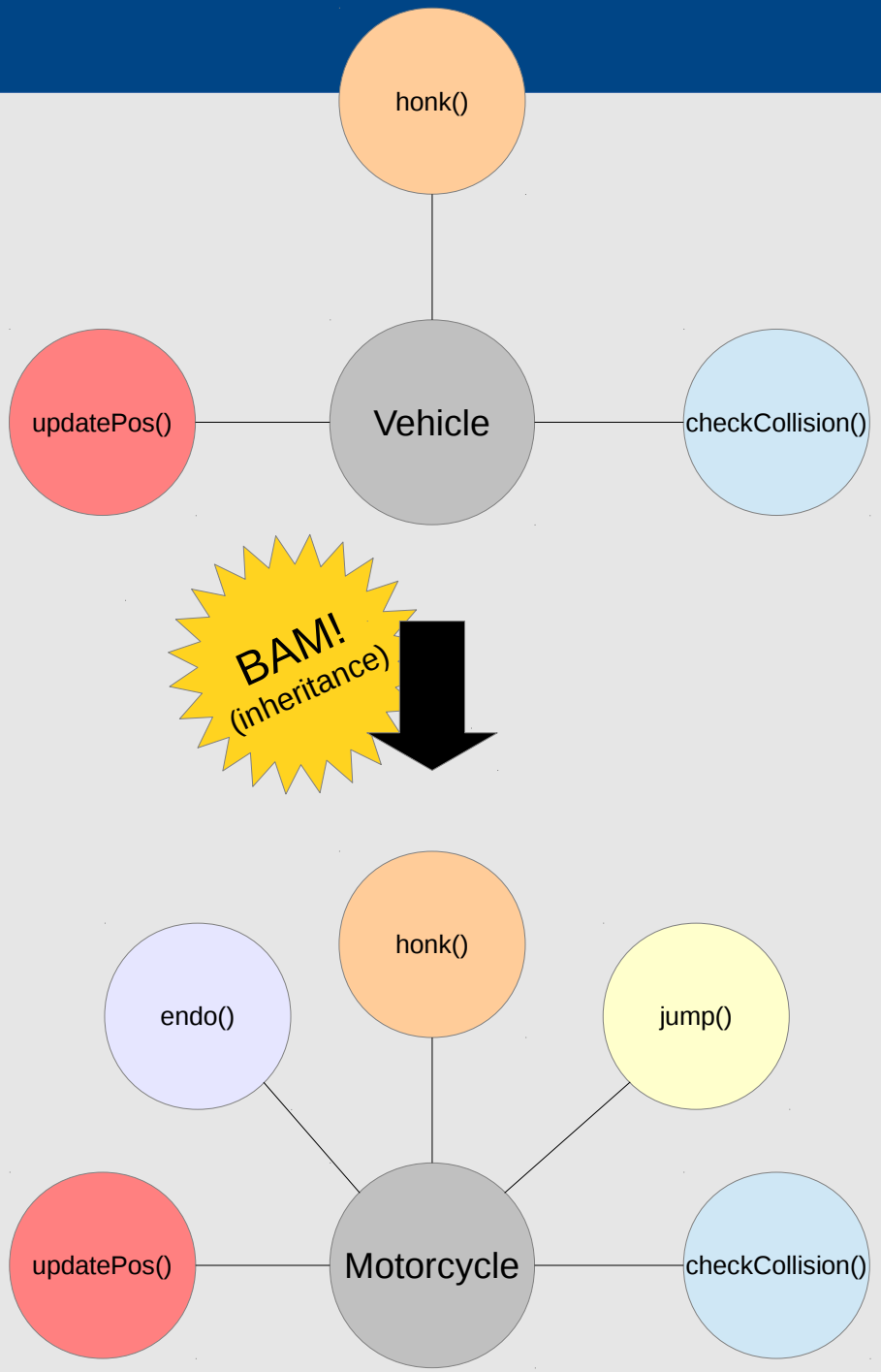
Polymorphism



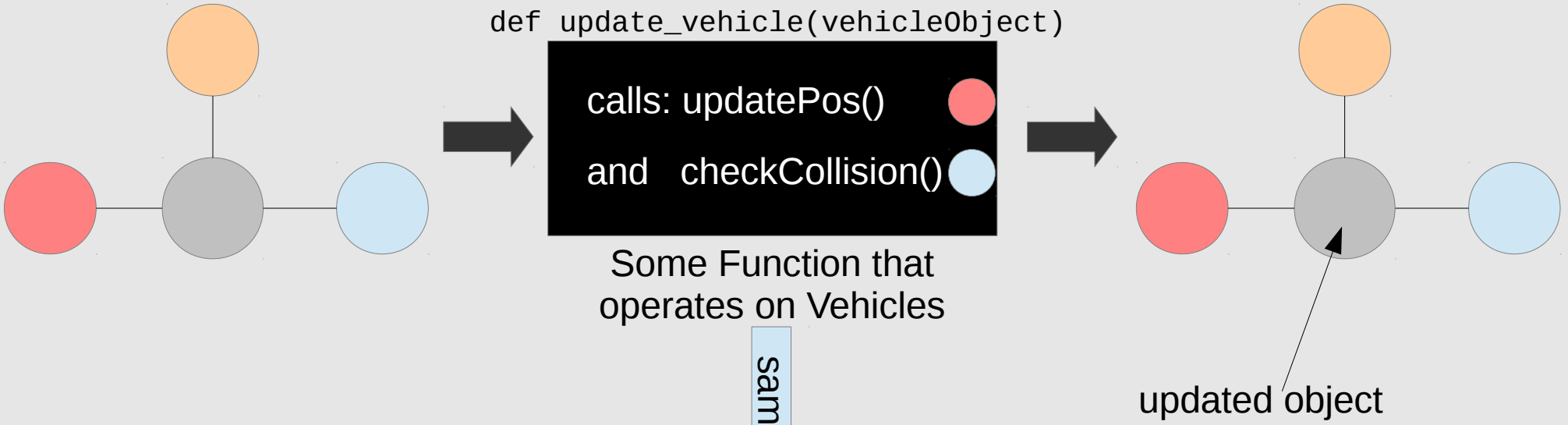
Polymorphism



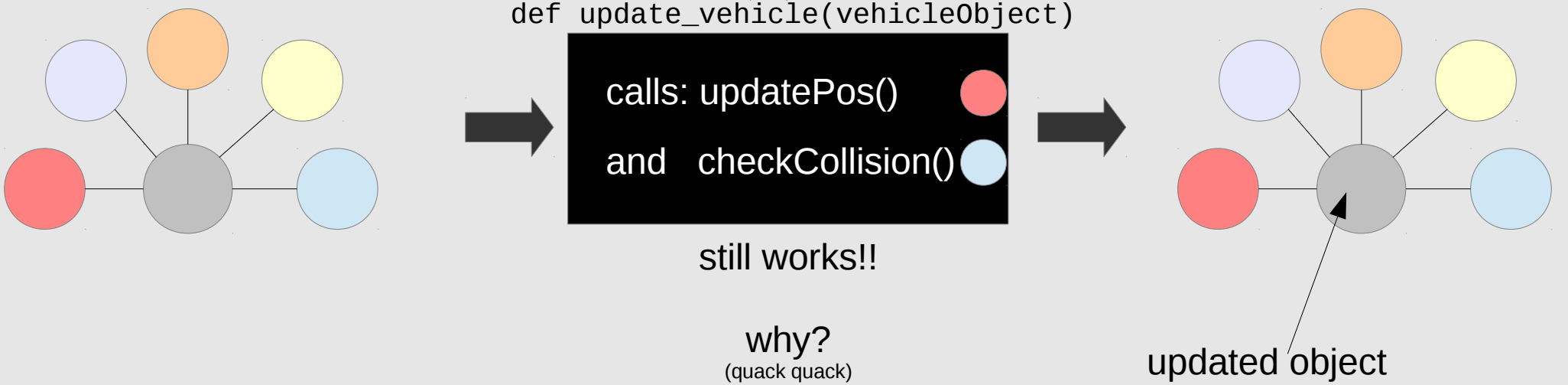
Polymorphism



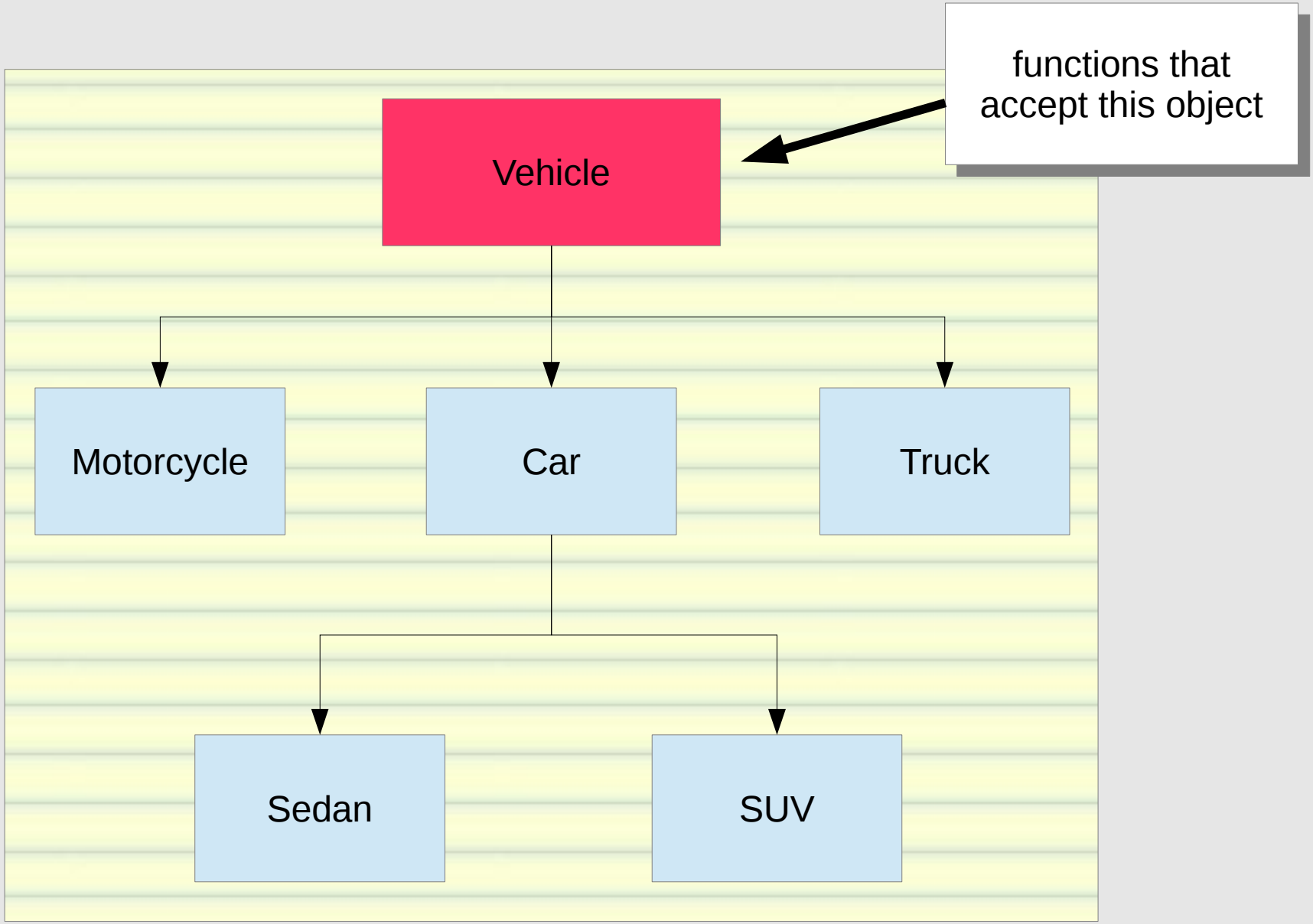
Polymorphism



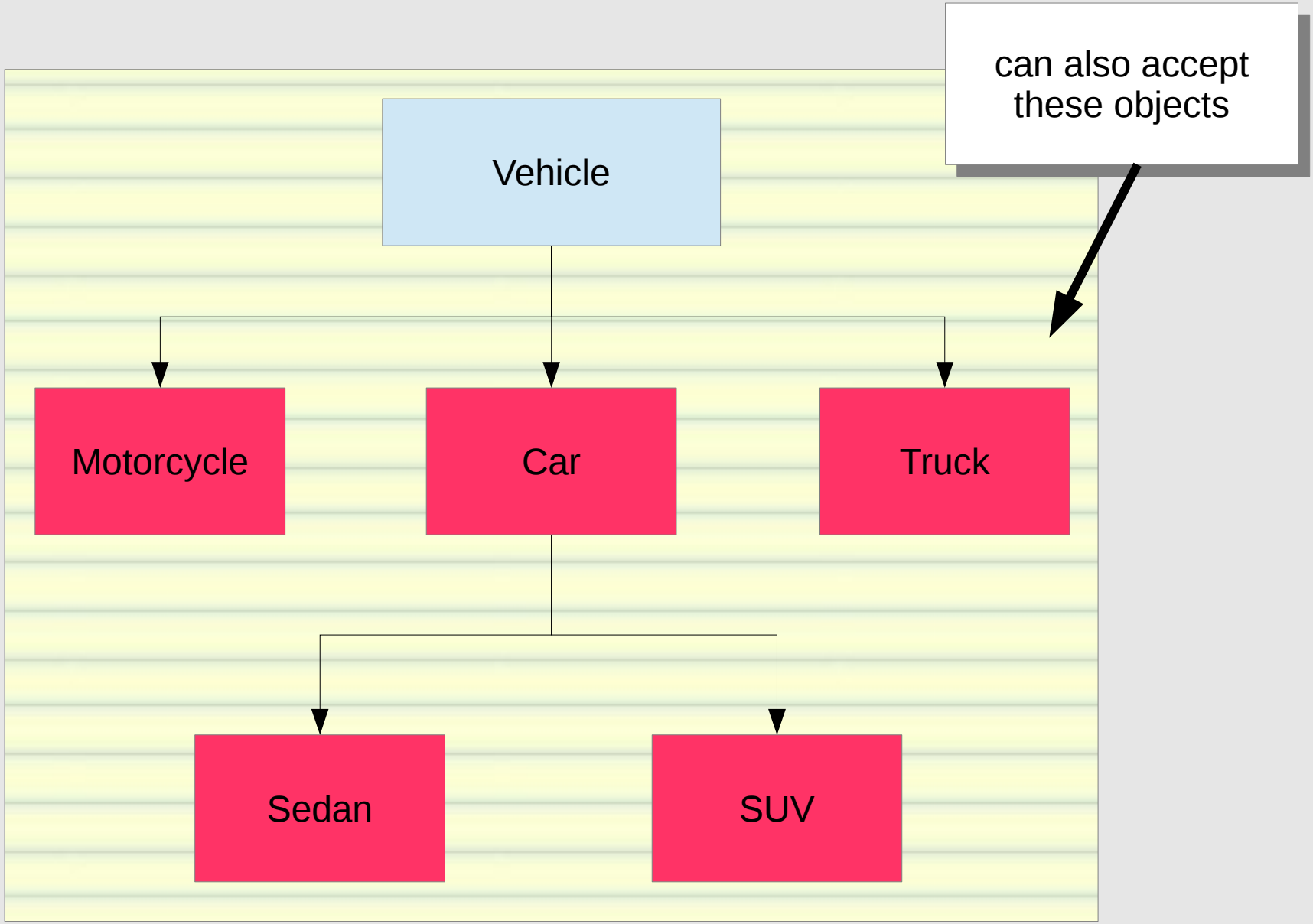
same function!



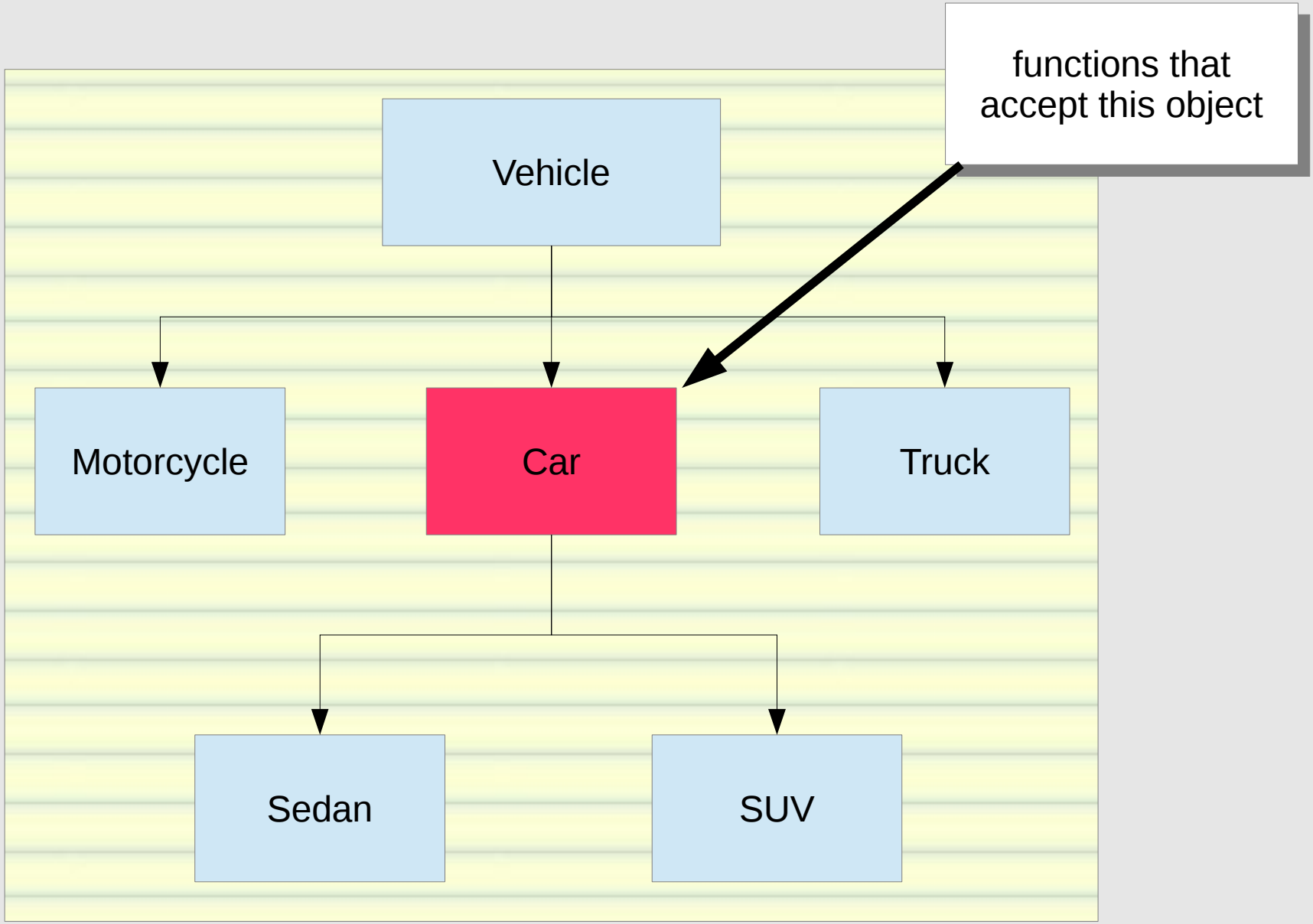
Polymorphism



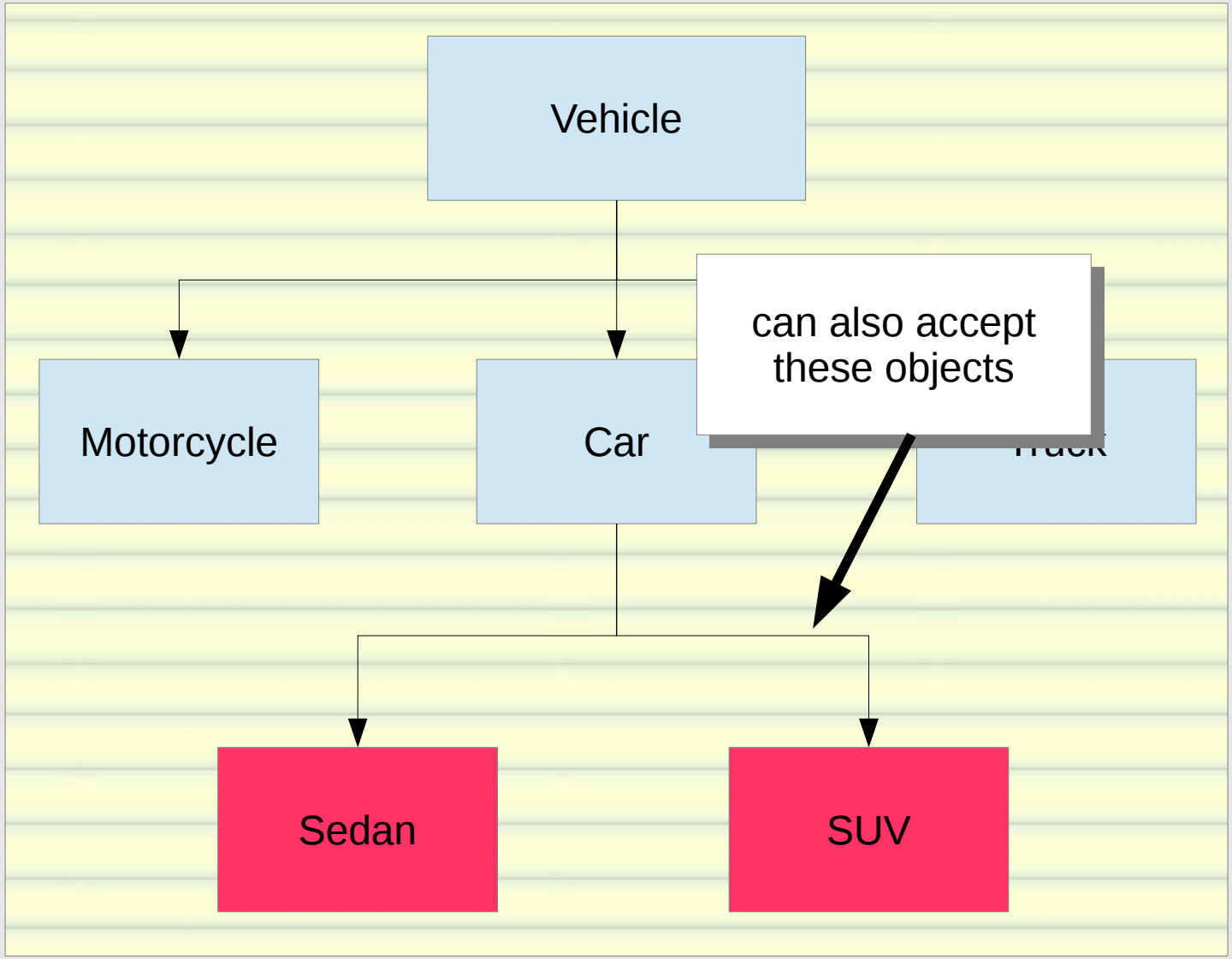
Polymorphism



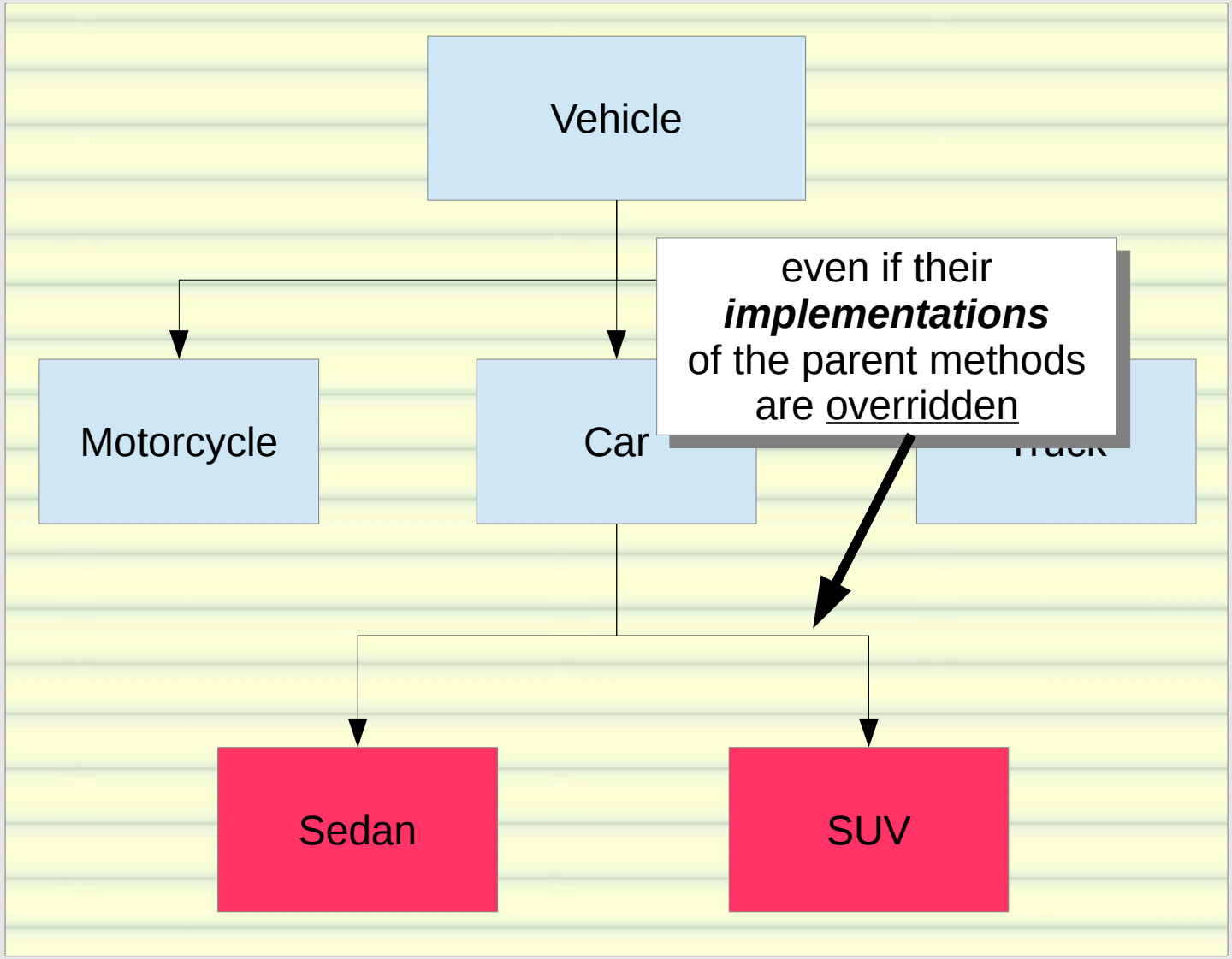
Polymorphism



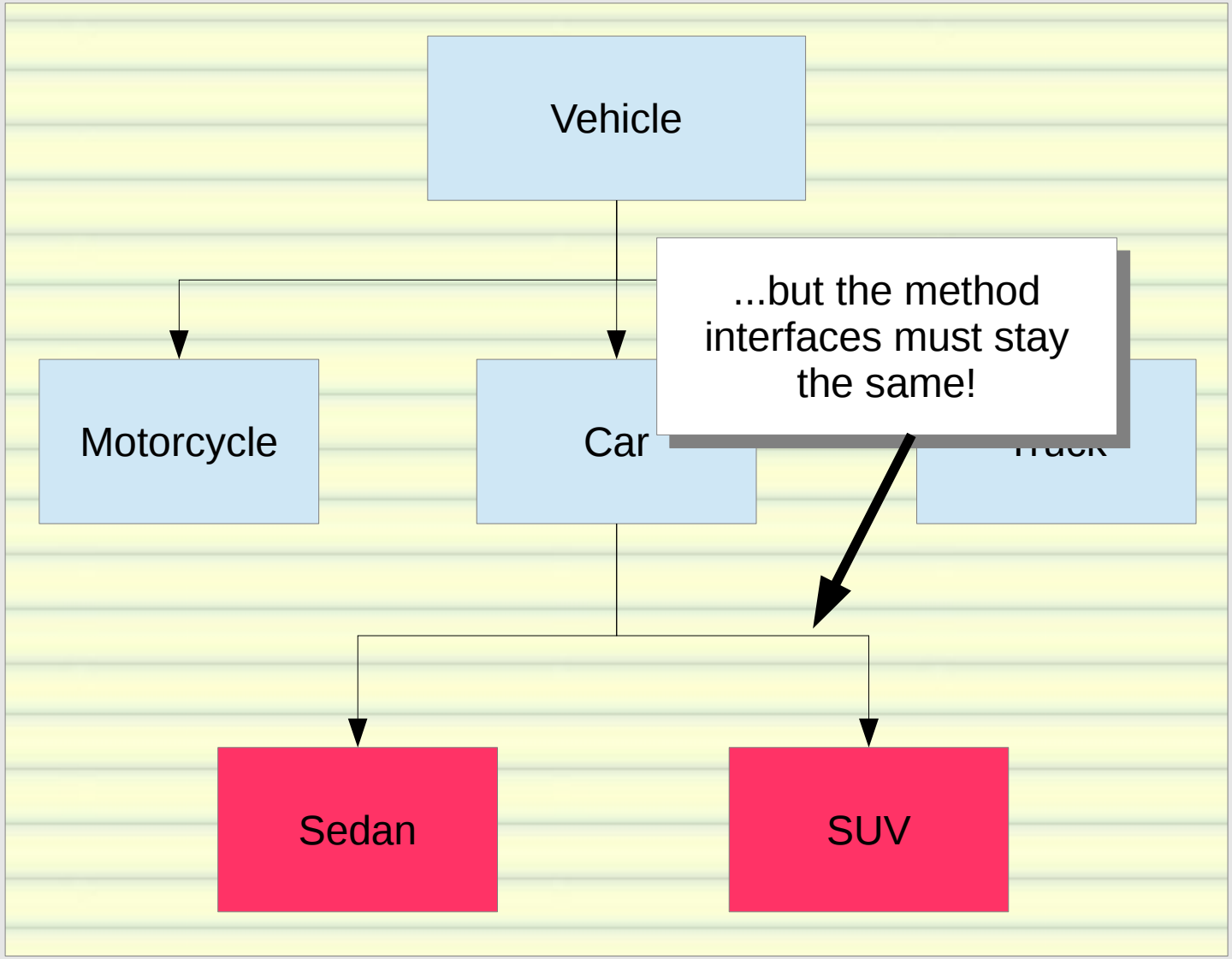
Polymorphism



Polymorphism



Polymorphism



Polymorphism

```
1  class Vehicle(object):
2      def __init__(self, numberOfTires):
3          self._numberOfTires = numberOfTires
4
5      def getNumberOfTires(self):
6          return self._numberOfTires
7
8      def setNumberOfTires(self, numberOfTires):
9          self._numberOfTires = numberOfTires
10
11     def getDescription(self):
12         return "A vehicle with %i tires" % self._numberOfTires
13
14
15 class Car(Vehicle):
16     def __init__(self):
17         super(Car, self).__init__(4)
18         self._plateNumber = None
19
20     def setLicensePlate(self, plateNumber):
21         self._plateNumber = plateNumber
22
23     def getDescription(self):
24         return "A CAR with %i tires" % self._numberOfTires
25
26
27 def print_description(foo):
28     print foo.getDescription()
29
30 my_18_wheeler = Vehicle(18)
31 my_car = Car()
32
33 print_description(my_18_wheeler)
34 print_description(my_car)
```

Polymorphism

Parent Class

```
1  class Vehicle(object):
2      def __init__(self, numberOfTires):
3          self._numberOfTires = numberOfTires
4
5      def getNumberOfTires(self):
6          return self._numberOfTires
7
8      def setNumberOfTires(self, numberOfTires):
9          self._numberOfTires = numberOfTires
10
11     def getDescription(self):
12         return "A vehicle with %i tires" % self._numberOfTires
13
14
15 class Car(Vehicle):
16     def __init__(self):
17         super(Car, self).__init__(4)
18         self._plateNumber = None
19
20     def setLicensePlate(self, plateNumber):
21         self._plateNumber = plateNumber
22
23     def getDescription(self):
24         return "A CAR with %i tires" % self._numberOfTires
25
26
27 def print_description(foo):
28     print foo.getDescription()
29
30 my_18_wheeler = Vehicle(18)
31 my_car = Car()
32
33 print_description(my_18_wheeler)
34 print_description(my_car)
```

Polymorphism

Child Class

```
1  class Vehicle(object):
2      def __init__(self, numberOfTires):
3          self._numberOfTires = numberOfTires
4
5      def getNumberOfTires(self):
6          return self._numberOfTires
7
8      def setNumberOfTires(self, numberOfTires):
9          self._numberOfTires = numberOfTires
10
11     def getDescription(self):
12         return "A vehicle with %i tires" % self._numberOfTires
13
14
15 class Car(Vehicle):
16     def __init__(self):
17         super(Car, self).__init__(4)
18         self._plateNumber = None
19
20     def setLicensePlate(self, plateNumber):
21         self._plateNumber = plateNumber
22
23     def getDescription(self):
24         return "A CAR with %i tires" % self._numberOfTires
25
26
27 def print_description(foo):
28     print foo.getDescription()
29
30 my_18_wheeler = Vehicle(18)
31 my_car = Car()
32
33 print_description(my_18_wheeler)
34 print_description(my_car)
```

Polymorphism

overridden method

```
1  class Vehicle(object):
2      def __init__(self, numberOfTires):
3          self._numberOfTires = numberOfTires
4
5      def getNumberOfTires(self):
6          return self._numberOfTires
7
8      def setNumberOfTires(self, numberOfTires):
9          self._numberOfTires = numberOfTires
10
11     def getDescription(self):
12         return "A vehicle with %i tires" % self._numberOfTires
13
14
15 class Car(Vehicle):
16     def __init__(self):
17         super(Car, self).__init__(4)
18         self._plateNumber = None
19
20     def setLicensePlate(self, plateNumber):
           self._plateNumber = plateNumber
22
23     def getDescription(self):
24         return "A CAR with %i tires" % self._numberOfTires
25
26
27 def print_description(foo):
28     print foo.getDescription()
29
30 my_18_wheeler = Vehicle(18)
31 my_car = Car()
32
33 print_description(my_18_wheeler)
34 print_description(my_car)
```

Polymorphism

```
1  class Vehicle(object):
2      def __init__(self, numberOfTires):
3          self._numberOfTires = numberOfTires
4
5      def getNumberOfTires(self):
6          return self._numberOfTires
7
8      def setNumberOfTires(self, numberOfTires):
9          self._numberOfTires = numberOfTires
10
11     def getDescription(self):
12         return "A vehicle with %i tires" % self._numberOfTires
13
14
15 class Car(Vehicle):
16     def __init__(self):
17         super(Car, self).__init__(4)
18         self._plateNumber = None
19
20     def setLicensePlate(self, plateNumber):
21         self._plateNumber = plateNumber
22
23     def getDescription(self):
24         return "A CAR with %i tires" % self._numberOfTires
25
26
27 def print_description(foo):
28     print foo.getDescription()
29
30 my_18_wheeler = Vehicle(18)
31 my_car = Car()
32
33 print_description(my_18_wheeler)
34 print_description(my_car)
```

polymorphic function

Can Accept:
- **Vehicle**
- **Car**
- **anything with a getDescription() method**

Polymorphism

```
1  class Vehicle(object):
2      def __init__(self, numberOfTires):
3          self._numberOfTires = numberOfTires
4
5      def getNumberOfTires(self):
6          return self._numberOfTires
7
8      def setNumberOfTires(self, numberOfTires):
9          self._numberOfTires = numberOfTires
10
11     def getDescription(self):
12         return "A vehicle with %i tires" % self._numberOfTires
13
14
15 class Car(Vehicle):
16     def __init__(self):
17         super(Car, self).__init__(4)
18         self._plateNumber = None
19
20     def setLicensePlate(self, plateNumber):
21         self._plateNumber = plateNumber
22
23     def getDescription(self):
24         return "A CAR with %i tires" % self._numberOfTires
25
26
27 def print_description(foo):
28     print foo.getDescription()
29
30 my_18_wheeler = Vehicle(18)
31 my_car = Car()
32
33 print_description(my_18_wheeler)
34 print_description(my_car)
```

Output:

```
A vehicle with 18 tires
A CAR with 4 tires
```

Abstract Base Classes

Sometimes a method *should* exist

BUT

its implementation isn't know until we know
the child class implementation

Abstract Base Classes

```
class Account(object):
    def __init__(self):
        self._balance = 0.0

    def withdraw(self, amount):
        if self._balance - amount >= 0.0:
            self._balance -= amount
            return True
        else:
            return False

    def deposit(self, amount):
        self._balance += amount

    def deductFees(self):
        raise NotImplementedError
```

This is how to raise (i.e. throw) an exception!

Abstract Base Classes

```
class Account(object):
    def __init__(self):
        self._balance = 0.0

    def withdraw(self, amount):
        if self._balance - amount >= 0.0:
            self._balance -= amount
            return True
        else:
            return False

    def deposit(self, amount):
        self._balance += amount

    def deductFees(self):
        raise NotImplementedError
```

This is how to raise (i.e. throw) an exception!

!! RECALL !!

From ECE-203

We learned how to ***catch*** exceptions.

```
1  my_list = range(5)
2
3  print 'for-loop:'
4  for i in my_list:
5      print i
6
7  print 'raw iter() w/ try-except:'
8  iterator = iter(my_list)
9  while True:
10     try:
11         i = iterator.next()
12     except:
13         break
14     else:
15         print i
16
17 del iterator
```

Abstract Base Classes

```
class Account(object):
    def __init__(self):
        self._balance = 0.0

    def withdraw(self, amount):
        if self._balance - amount >= 0.0:
            self._balance -= amount
            return True
        else:
            return False

    def deposit(self, amount):
        self._balance += amount

    def deductFees(self):
        raise NotImplementedError
```

```
class Checking(Account):
    def __init__(self):
        super(Checking, self).__init__()
        self._badCheckFee = 5.00
        self._lowBalanceFee = 25.00

    def processCheck(self, amount):
        if not self.withdraw(amount):
            self._balance -= self._badCheckFee

    def deductFees(self):
        if self._balance < 200.00:
            self._balance -= self._lowBalanceFee
```

Abstract Base Classes

```
class Account(object):
    def __init__(self):
        self._balance = 0.0

    def withdraw(self, amount):
        if self._balance - amount >= 0.0:
            self._balance -= amount
            return True
        else:
            return False

    def deposit(self, amount):
        self._balance += amount

    def deductFees(self):
        raise NotImplementedError
```

```
class Checking(Account):
    def __init__(self):
        super(Checking, self).__init__()
        self._badCheckFee = 5.00
        self._lowBalanceFee = 25.00

    def processCheck(self, amount):
        if not self.withdraw(amount):
            self._balance -= self._badCheckFee

    def deductFees(self):
        if self._balance < 200.00:
            self._balance -= self._lowBalanceFee
```

```
def monthly_account_update(account):
    account.deductFees()
```

Abstract Base Classes

```
class Account(object):
    def __init__(self):
        self._balance = 0.0

    def withdraw(self, amount):
        if self._balance - amount >= 0.0:
            self._balance -= amount
            return True
        else:
            return False

    def deposit(self, amount):
        self._balance += amount

    def deductFees(self):
        raise NotImplementedError
```

```
class Checking(Account):
    def __init__(self):
        super(Checking, self).__init__()
        self._badCheckFee = 5.00
        self._lowBalanceFee = 25.00

    def processCheck(self, amount):
        if not self.withdraw(amount):
            self._balance -= self._badCheckFee

    def deductFees(self):
        if self._balance < 200.00:
            self._balance -= self._lowBalanceFee
```

```
def monthly_account_update(account):
    account.deductFees()
```

```
if __name__ == '__main__':
    someAccount = Checking()
    someAccount.deposit(300)
    someAccount.processCheck(120)
    monthly_account_update(someAccount)
```

Abstract Base Classes

```
class Account(object):
    def __init__(self):
        self._balance = 0.0

    def withdraw(self, amount):
        if self._balance - amount >= 0.0:
            self._balance -= amount
            return True
        else:
            return False

    def deposit(self, amount):
        self._balance += amount

    def deductFees(self):
      3 raise NotImplementedError
```

```
class Checking(Account):
    def __init__(self):
        super(Checking, self).__init__()
        self._badCheckFee = 5.00
        self._lowBalanceFee = 25.00

    def processCheck(self, amount):
        if not self.withdraw(amount):
            self._balance -= self._badCheckFee

    def deductFees(self):
        if self._balance < 200.00:
            self._balance -= self._lowBalanceFee
```

```
def monthly_account_update(account):
  2 account.deductFees()
```

```
if __name__ == '__main__':
    someAccount = Checking()
    someAccount.deposit(300)
    someAccount.processCheck(120)
  1 monthly_account_update(someAccount)
```

```
Traceback (most recent call last):
  File "account.py", line 34, in <module>
  1 monthly_account_update(someAccount)
  File "account.py", line 30, in monthly_account_update
  2 account.deductFees()
  File "account.py", line 16, in deductFees
  3 raise NotImplementedError
NotImplementedError
```

Using Dictionaries of Functions

i.e. Python doesn't have a switch-case statement

DESIGN PATTERN: Dictionaries of Function References

Python does not have a switch-case construct like C/Java

```
int ans;
switch (operation) {
    case "add":
        ans = x + y;
        break;
    case "sub":
        ans = x - y;
        break;
    case "mul":
        ans = x * y;
        break;
    case "div":
        ans = x / y;
        break;
    default:
        ans = 0;
}
```

DESIGN PATTERN: Dictionaries of Function References

Solution:
Use a dictionary of function references

Issue:
Can be inconvenient to write a a function for each case!!

```
1  def add (x, y):
2      return x + y
3
4  def sub (x, y):
5      return x - y
6
7  def mul (x, y):
8      return x * y
9
10 def div (x, y):
11     return x / y
12
13
14 mops = {
15     "add": add,
16     "sub": sub,
17     "mul": mul,
18     "div": div
19 }
20
21
22 def do_math(operation, x, y):
23     if not operation in mops:
24         ans = 0
25     else:
26         ans = mops[operation](x, y)
27
28     print "%s %i, %i \t-> %i" % (operation, x, y, ans)
29
30 do_math ("add", 1, 2)
```

Output:

```
add 1, 2        -> 3
```

DESIGN PATTERN: Dictionaries of Function References

Issue:
Can be inconvenient to write a function for each case!!

Solution:
Lambda Functions
(i.e. nameless functions)

```
 1 def add_numbers(x, y):
 2     return x + y
 3
 4
 5 foo = add_numbers
 6 bar = lambda x, y: x + y
 7
 8
 9 print foo(500, 20)
10 print bar(500, 20)
11
12
13 # Output:
14 # 520
15 # 520
```

DESIGN PATTERN: Dictionaries of Function References

Issue:
Can be inconvenient to write a function for each case!!

Solution:
Lambda Functions
(i.e. nameless functions)

```
 1 def add_numbers(x, y):
 2     return x + y
 3
 4
 5 foo = add_numbers
 6 bar = lambda x, y: x + y
 7
 8
 9 print foo(500, 20)
10 print bar(500, 20)
11
12
13 # Output:
14 # 520
15 # 520
```

DESIGN PATTERN: Dictionaries of Function References

Issue:
Can be inconvenient to write a function for each case!!

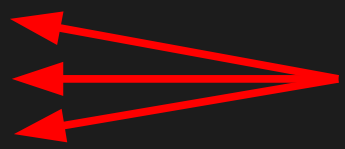
Solution:
Lambda Functions
(i.e. nameless functions)

```
 1 def add_numbers(x, y):
 2     return x + y
 3
 4
 5 foo = add_numbers
 6 bar = lambda x, y: x + y
 7
 8
 9 print foo(500, 20)
10 print bar(500, 20)
11
12
13 # Output:
14 # 520
15 # 520
```

DESIGN PATTERN: Dictionaries of Function References

Issue:
Can be inconvenient to write a function for each case!!

Solution:
Lambda Functions
(i.e. nameless functions)



```
mops = {
    "add": lambda x, y: x + y,
    "sub": lambda x, y: x - y,
    "mul": lambda x, y: x * y,
    "div": lambda x, y: x / y
}

def do_math(operation, x, y):
    if not operation in mops:
        ans = 0
    else:
        ans = mops[operation](x, y)

    print "%s %i, %i \t-> %i" % (operation, x, y, ans)

do_math ("add", 1, 2)
do_math ("sub", 50, 2)
do_math ("mul", 5, 5)
do_math ("nop", 100, 23)

# Output:
#add 1, 2 -> 3
#sub 50, 2 -> 48
#mul 5, 5 -> 25
#nop 100, 23 -> 0
```

DESIGN PATTERN: Dictionaries of Function References

Issue:
Messy Way to Check!!

Solution:
Lambda Functions
(Yes, AGAIN!!)

```
1  mops = {
2      "add": lambda x, y: x + y,
3      "sub": lambda x, y: x - y,
4      "mul": lambda x, y: x * y,
5      "div": lambda x, y: x / y
6  }
7
8  def do_math(operation, x, y):
9      if not operation in mops:
10         ans = 0
11     else:
12         ans = mops[operation](x, y)
13
14     print "%s %i, %i \t-> %i" % (operation, x, y, ans)
15
16 do_math ("add", 1, 2)
17 do_math ("sub", 50, 2)
18 do_math ("mul", 5, 5)
19 do_math ("nop", 100, 23)
20
21 # Output:
22 #add 1, 2 -> 3
23 #sub 50, 2 -> 48
24 #mul 5, 5 -> 25
25 #nop 100, 23 -> 0
```

DESIGN PATTERN: Dictionaries of Function References

```
1  mops = {
2      "add": lambda x, y: x + y,
3      "sub": lambda x, y: x - y,
4      "mul": lambda x, y: x * y,
5      "div": lambda x, y: x / y
6  }
7
8  def do_math(operation, x, y):
9      foo = mops.get(operation, lambda x, y: 0)
10
11     ans = foo (x, y);
12     print "%s %i, %i \t-> %i" % (operation, x, y, ans)
13
14 do_math ("add", 1, 2)
15 do_math ("sub", 50, 2)
16 do_math ("mul", 5, 5)
17 do_math ("nop", 100, 23)
18
19 # Output:
20 #add 1, 2 -> 3
21 #sub 50, 2 -> 48
22 #mul 5, 5 -> 25
23 #nop 100, 23 -> 0
```

returned if **operation** isn't a key in the dictionary

DESIGN PATTERN: Dictionaries of Function References

```
1  mops = {
2      "add": lambda x, y: x + y,
3      "sub": lambda x, y: x - y,
4      "mul": lambda x, y: x * y,
5      "div": lambda x, y: x / y
6  }
7
8  def do_math(operation, x, y):
9      foo = mops.get(operation, lambda x, y: 0)
10
11     ans = foo (x, y);
12     print "%s %i, %i \t-> %i" % (operation, x, y, ans)
13
14 do_math ("add", 1, 2)
15 do_math ("sub", 50, 2)
16 do_math ("mul", 5, 5)
17 do_math ("nop", 100, 23)
18
19 # Output:
20 #add 1, 2 -> 3
21 #sub 50, 2 -> 48
22 #mul 5, 5 -> 25
23 #nop 100, 23 -> 0
```

we can combine these two steps

DESIGN PATTERN: Dictionaries of Function References

```
1  mops = {
2      "add": lambda x, y: x + y,
3      "sub": lambda x, y: x - y,
4      "mul": lambda x, y: x * y,
5      "div": lambda x, y: x / y
6  }
7
8  def do_math(operation, x, y):
9      ans = mops.get(operation, lambda x, y: 0) (x, y)
10
11     print "%s %i, %i \t-> %i" % (operation, x, y, ans)
12
13 do_math ("add", 1, 2)
14 do_math ("sub", 50, 2)
15 do_math ("mul", 5, 5)
16 do_math ("nop", 100, 23)
17
18 # Output:
19 #add 1, 2 -> 3
20 #sub 50, 2 -> 48
21 #mul 5, 5 -> 25
22 #nop 100, 23 -> 0
```

DESIGN PATTERN: Dictionaries of Function References

```
1  mops = {
2      "add": lambda x, y: x + y,
3      "sub": lambda x, y: x - y,
4      "mul": lambda x, y: x * y,
5      "div": lambda x, y: x / y
6  }
7
8  def do_math(operation, x, y):
9      ans = mops.get(operation, lambda x, y: 0) (x, y)
10
11     print "%s %i, %i \t-> %i" % (operation, x, y, ans)
12
13 do_math ("add", 1, 2)
14 do_math ("sub", 50, 2)
15 do_math ("mul", 5, 5)
16 do_math ("nop", 100, 23)
17
18 # Output:
19 #add 1, 2 -> 3
20 #sub 50, 2 -> 48
21 #mul 5, 5 -> 25
22 #nop 100, 23 -> 0
```