CSCI 553: Networking III

Unix Network Programming

Spring 2007

Object Oriented Programming

- A Word about the Test
  - Books, Notes & Account
  - Assigned Seats
  - Study Questions
- Objects & Testing
  - Set up .bashrc for JUnit usage
- Lab 05
  - Finish 5.1 before leaving

Introduction

- Suppose you want to simulate a small ecosystem, such as a tidal pool, that contains many different kinds of things
  - Plants (don't move)
  - Fish (swim in three dimensions)
  - Crawly things (cling to the surface most of the time)
- The procedural way to do it uses a type-switch

```
for time in simulation_period:
    for thing in world:
        if type(thing) is plant:
            update_plant(thing, time)
        elif type(thing) is fish:
            update_fish(thing, time)
        elif type(thing) is creepy_crawly:
            update_creepy_crawly(thing, time)
```

- But:
  - Every time you add a new type of thing, you have to find and update all the type-switches
  - It's very easy to make a cut-and-paste mistake

Objects to the Rescue

- Object-oriented programming (OOP) solves both problems
  - Each object knows how to update itself

```
for time in simulation_period:
    for thing in world:
        thing.update(time)
```

  - Don't have to change old code when adding new types of things
  - No chance of calling the wrong function
- Seems like a small change, but it allows programmers to think and design at a higher level
  - Also allows them to make more powerful mistakes…
  - Ideas apply to all modern languages
  - But there's more variation in form than there is with loops, conditionals, and functions

- Modern languages encourage programmers to define abstract data types (ADTs)
  - “Abstract” because they hide the details of their implementation
- Programmers interact with them through a limited set of operations, rather than by manipulating data directly
  - Fewer things can go wrong
  - Easier to read resulting code
  - Makes code easier to maintain, since internals can be changed without changing calling code

Classes and Instances

- An ADT is usually created by defining a class that specifies:
  - How the ADT stores state (its member variables)
  - What it can do (its methods)
  - And yes, classes are also objects, just like functions
- Programmers then create objects that are instances of that class
  - Each object of a particular ADT shares the class's methods, but has its own members
  - So changes to one object do not affect the state of others

Classes and Instances

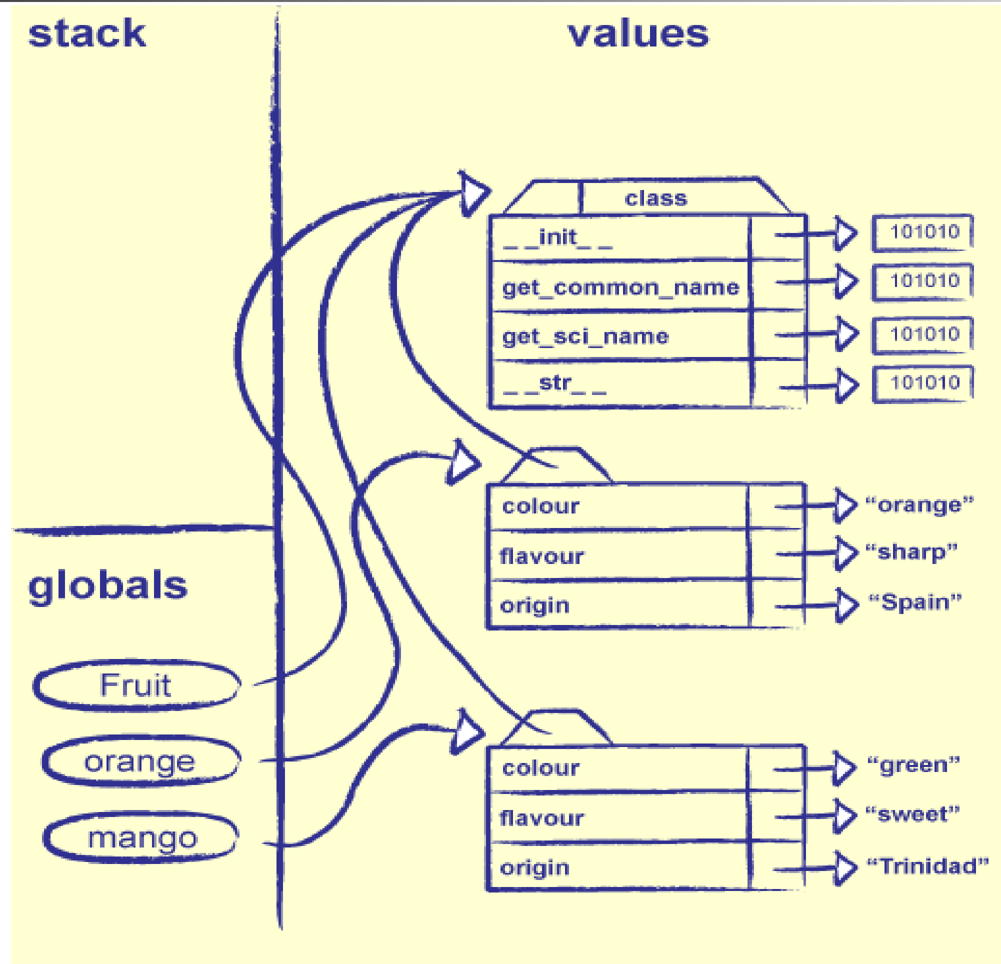


Figure 1: Memory Model for Classes and Objects

Defining a Class

- Define a class in Python using the class keyword
  - Name of the new class is usually followed by object in parentheses
    - We'll see other options later
  - Then ":" and an indented block containing the class's contents

```
class Empty(object):
    pass
```

  - pass means “do nothing”, i.e., create an empty class
    - Not particularly useful, but you have to start somewhere

Creating an Instance

- Create a new instance of the class by calling the class name as if it were a function

```
if __name__ == '__main__':
      first = Empty()
      second = Empty()
      print 'first has id', id(first)
      print 'second has id', id(second)
```

first has id 5086860

second has id 5086892

- id returns the object's hash code
  - Doesn't mean anything: just distinguishes objects
- Note how main body of program is put in a block under if __name__ == '__main__':
  - Otherwise, it will be executed when other programs import the class

Methods

- Give the class methods by defining functions inside it
- The object itself is always passed to the method as its first argument
  - Universally called self
  - Unlike this in C++ and Java, the name is just a convention
  - But everyone uses it, and you should too
- object.method(argument) is equivalent to:
  - Find the class C that object is an instance of
  - Call C.method(object, argument)

```
class Greeting(object):
    def say(self, name):
        print 'Hello, %s!' % name

if __name__ == '__main__':
    greet = Greeting()
    greet.say('object')

Hello, object!
```

Creating Members

- Every object is a new scope for variable names
  - Just like a module, or a function call
- The values in an object's scope are its members
  - Create members use dotted notation: self.x = 3
  - Gives the current object a new member x with the value 3
  - Or overwrites the existing member x with the value 3

Creating Members

```
class Point(object):
    def set_values(self, x, y):
        self.x = x
        self.y = y
    def get_values(self):
        return (self.x, self.y)
    def norm(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)

if __name__ == '__main__':
    p = Point()
    p.set_values(1.2, 3.5)
    print 'p is', p.get_values()
    print 'norm is', p.norm()
```

p is (1.2, 3.5)

norm is 3.7

Creating Members

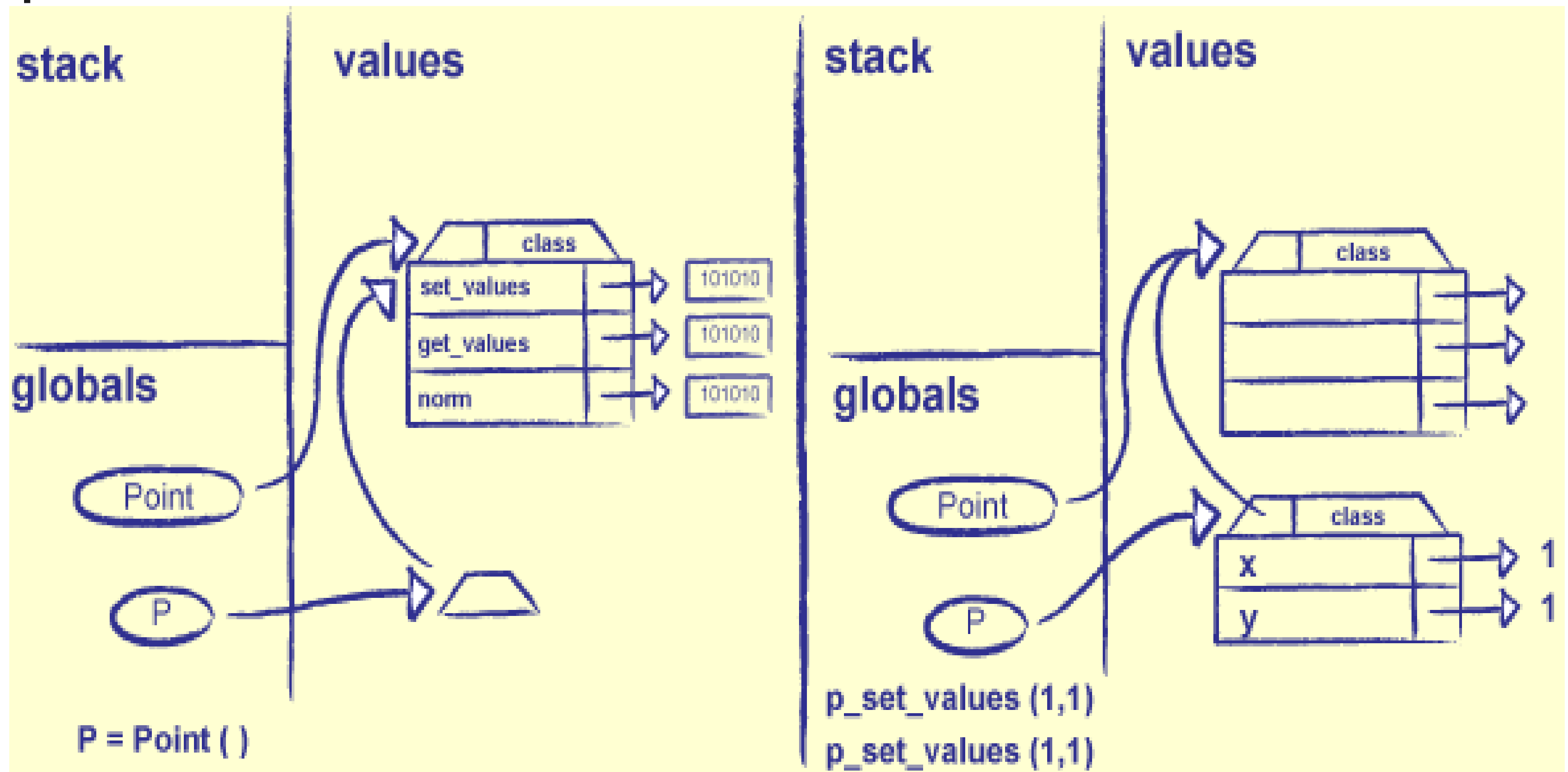


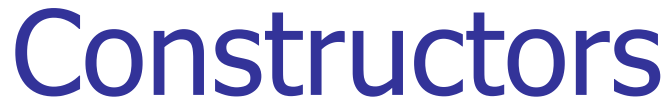
Figure 2: Creating a Simple Point

Encapsulation

- Encapsulation is one of the three defining principles of OOP
  - Programs are much easier to write, read, and maintain if object members are only ever accessed by methods
- But unlike C++, Java, and C#, Python doesn't allow programmers to hide methods or data members

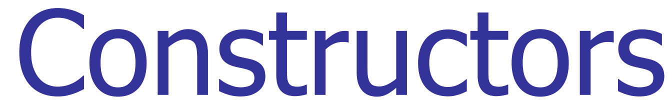
```
p = Point()
p.x = 3.5
p.y = 4.25
print 'point is', p.get_values()
point is (3.5, 4.25)
```

- Any function or method can see and modify any object's internals
  - Resist the temptation to program this way!
  - If you manipulate an object's internals directly, you have to change your program when you change the object's implementation



Constructors

- If a class has a method called __init__, Python will call it when building new instances
  - Hence the name constructor
  - Simpler than creating a blank object, then initializing its members
  - And there's less chance the programmer will forget to do the initialization



Constructors

```
class Point(object):
    def __init__(self, x=0, y=0):
        self.reset(x, y)
    def reset(self, x, y):
        assert (type(x) is int) and (x >= 0), 'x is not non-negative integer'
        assert (type(y) is int) and (y >= 0), 'y is not non-negative integer'
        self.x = x
        self.y = y
    def get(self):
        return (self.x, self.y)
    def norm(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)

if __name__ == '__main__':
    p = Point(1, 1)
    print 'point is initially', p.get()
    p.reset(2, 2)
    print 'p moved to', p.get()
```

point is initially (1, 1)

p moved to (2, 2)

Constructor Style

- A class can only have one constructor
  - Some languages allow classes to have several, distinguished by argument types
  - But since Python doesn't use type declarations, this wouldn't work
- It's good style to create all of the object's members in the constructor
  - So that people only have to look in one place to find what members exist
- Note how the class checks values before changing the object's state
  - Remember: fail early, fail often

Special Methods

- __init__ is just one example of a special method
  - All have names beginning and ending with double underscore
  - Give programmers a way to make their data types look like those built into Python
- Most widely used is __str__
  - When Python needs a text representation of an object, it:
  - Calls __str__ if it exists, or
  - Creates a default representation that shows the object's location in memory

```
class Point(object):
    def __str__(self):
        return '(%4.2f, %4.2f)' % (self.x, self.y)

if __name__ == '__main__':
    p = Point(3, 4)
    print 'point is', p
```

```
point is (3, 4)
```

- Suppose we have a class Organism that represents living things
  - Common name, scientific name, …
- Want to create a class Mammal
  - Body temperature, gestation period, …
- Wrong: copy Organism's definition and add more members and methods
  - “Anything repeated in two or more places will eventually be wrong in at least one.”
- Right: use inheritance
  - The second defining principle of OOP
  - Derive a child class from a parent
  - The child has all the members and methods of its parents, plus whatever else we give it

Inheritance Example

```
class Organism(object):
    def __init__(self, common_name, sci_name):
        self.common_name = common_name
        self.sci_name = sci_name
    def get_common_name(self):
        return self.common_name
    def get_sci_name(self):
        return self.sci_name
    def __str__(self):
        return '%s (%s)' % (self.common_name, self.sci_name)
```

```
class Mammal(Organism):
    def __init__(self, common_name, sci_name, body_temp, gest_period):
        Organism.__init__(self, common_name, sci_name)
        self.body_temp = body_temp
        self.gest_period = gest_period
    def get_body_temp(self):
        return self.body_temp
    def get_gest_period(self):
        return self.gest_period
    def __str__(self):
        extra = ' %4.2f degrees / %d days' % (self.body_temp, self.gest_period)
        return Organism.__str__(self) + extra

if __name__ == '__main__':
    creature = Mammal('wolf', 'canis lupus', 38.7, 63)
    print creature
```

wolf (canis lupus) 38.70 degrees / 63 days

Inheritance Example

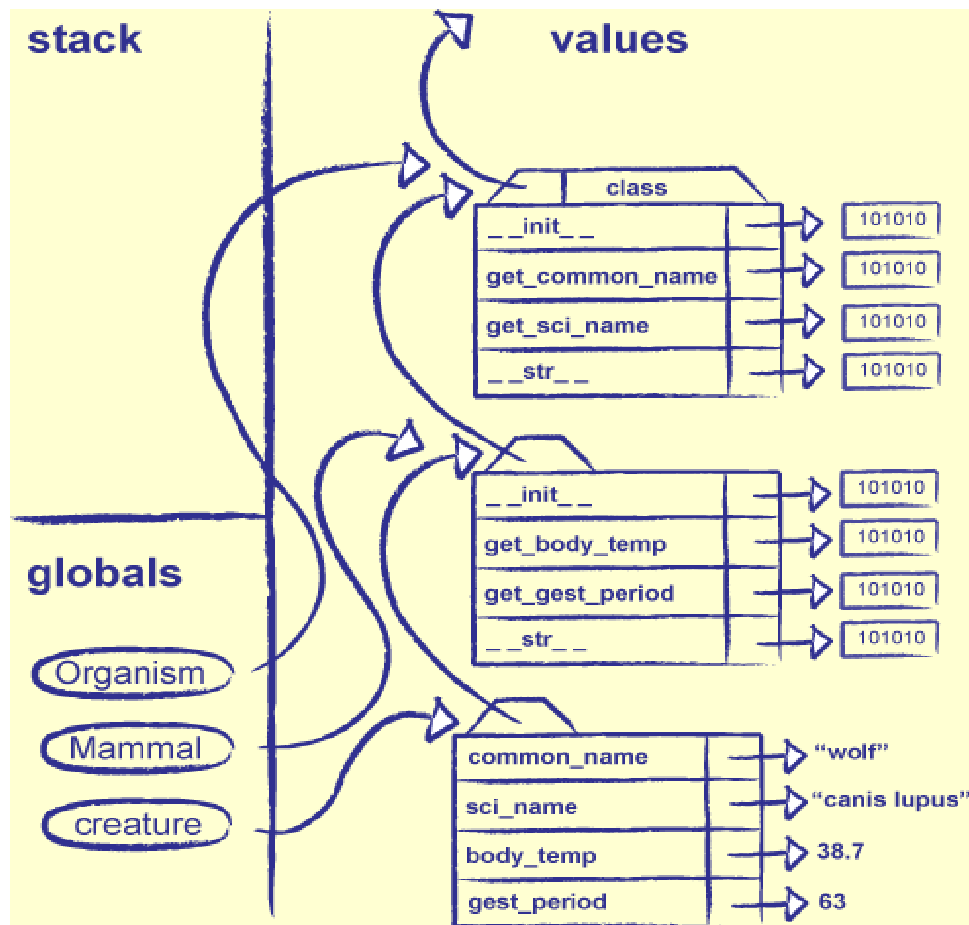


Figure 3: Memory Model for Inheritance

Overriding Methods

- Mammal's constructor calls Organism's to initialize the organism-ish bits of the object
- And Mammal defines its own __str__ method
  - Overrides the one defined by Organism
  - Mammal.__str__ calls Organism.__str__ for the same reason that Mammal.__init__ calls Organism.__init__
- Python always calls the most specific method
  - Keep the memory model in mind when figuring out what this will be

Polymorphism

- Polymorphism means "having more than one form"
  - In object-oriented programming, it means handling specific objects in generic ways
  - The third and final defining principle of OOP
- Derive a new class Bird from Organism
  - As long as it only uses common methods, a single piece of code can work with both mammals and bird

```
class Bird(Organism):
        def __init__(self, common_name, sci_name, incubate_period):
            Organism.__init__(self, common_name, sci_name)
            self.incubate_period = incubate_period
        def get_incubate_period(self):
            return self.incubate_period
        def __str__(self):
            extra = ' %d days' % self.incubate_period
            return Organism.__str__(self) + extra

if __name__ == '__main__':
        creatures = [
            Bird('loon', 'gavia immer', 27),
            Mammal('grizzly bear', 'ursus arctos horribilis', 38.0, 210)
        ]
        for c in creatures:
            print c
loon (gavia immer) 27 days
grizzly bear (ursus arctos horribilis) 38.00 degrees / 210 days
```

Duck Typing

- Most languages only permit polymorphism via inheritance
  - Lowest common ancestor of two classes defines how interchangeable they are
- In Python, any two classes that define the same set of methods can be used interchangeably
  - Duck typing: "If it walks like a duck, and quacks like a duck, it might as well be a duck."
- Allows you to create plug-in replacements for files, strings, and other classes after the fact
- But makes it harder to figure out exactly what can be used in place of what

```
class Mineral(object):
    def __init__(self, common_name, sci_name, formula):
        self.common_name = common_name
        self.sci_name = sci_name
        self.formula = formula
    def get_common_name(self):
        return self.common_name
    def get_sci_name(self):
        return self.sci_name
    def __str__(self):
        return '%s/%s: %s' % (self.common_name, self.sci_name, self.formula)

if __name__ == '__main__':
    things = [
        Mammal('arctic hare', 'Lepus arcticus', 40.1, 50),
        Mineral("fool's gold", 'iron pyrite', 'FeS2')
    ]
    for t in things:
        print t.get_common_name(), 'is', t.get_sci_name()
```

```
arctic hare is Lepus arcticus
fool's gold is iron pyrite
```

The Liskov Substitution Principle

- The Liskov Substitution Principle states that it must always be possible to use an instance of a child class in place of an instance of its parent
- Means that Child.meth may ignore some of Parent.meth's pre-conditions, but may not impose more
  - Equivalently, Child.meth accepts everything thatParent.meth did, and possibly more
  - So any code that could call Parent.meth correctly is guaranteed to call Child.meth correctly too
- And Child.meth must satisfy all the post-conditions of Parent.meth, and may impose more
  - Child.meth's possible output is a subset of Parent.meth's
  - And any code that works correctly on the output of Parent.meth will still work if given an instance of Child instead
- The same constraint applies when a class evolves over time

Tidal Pools Revisited

- How to represent the creatures in a tidal pool?
  - Each species is a class
  - Use inheritance to separate plants from animals
  - Derive both Plant and Animal from Organism
- How to handle movement?
  - Give Organism two methods: can_move and move
    - Plant.can_move() returns False
    - Plant.move() raises an exception
  - Give Organism one method: move
    - Plant.move() does nothing
- Second simplifies code
  - Uses polymorphism instead of a conditional
- On the other hand, the existence of Plant.move implies that plants can do something they can't
  - Can't really choose between them without knowing what the rest of the code needs
- Usually doesn't make sense to design one class on its own

Class, Responsibility, Collaborator

- Many programmers use CRC cards when designing OO systems
  - Stands for “class, responsibility, collaborator”
- Standard 3×5 index cards
  - Top is the class name
  - Left side is point-form description of what the class can do
  - Right side lists other classes that this one interacts with

Class, Responsibility, Collaborator

- Designed so that you won't take them too seriously
  - Lay them out on a table
  - Talk through your program's execution
  - Move cards around, scribble new responsibilities and collaborators on them
  - Create new cards as needed

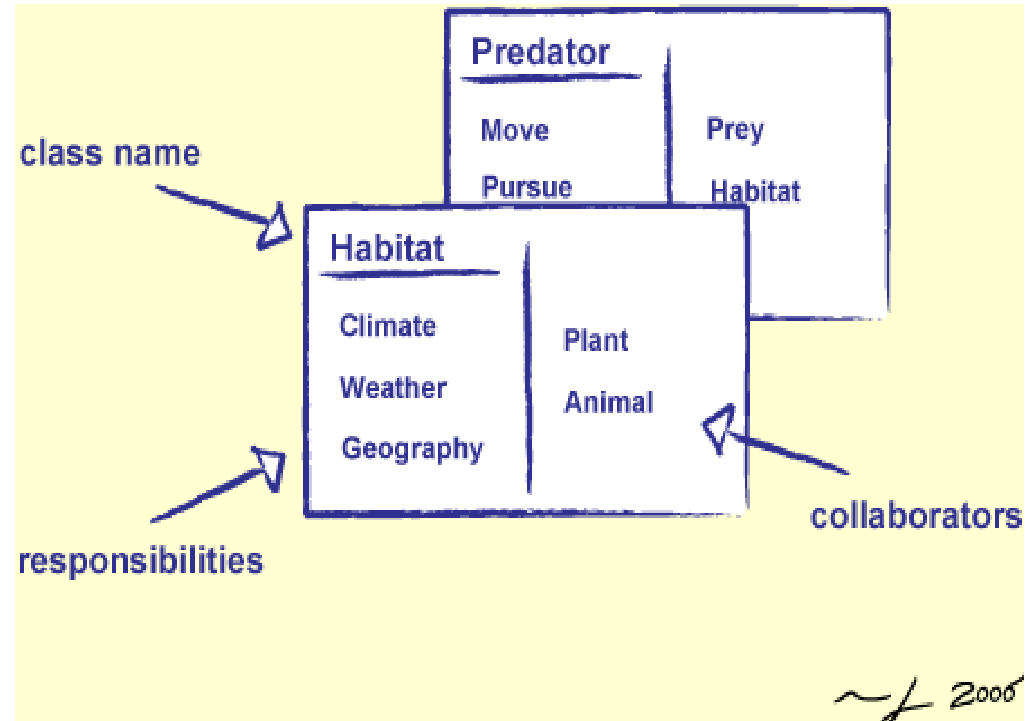


Figure 4: CRC Cards

Summary

- Classes and objects are just another way to modularize programs
  - But used well, they can make programs much simpler, and much more adaptable
- Remember: the goal is to simplify, not to dazzle