

# Object Fundamentals

---

Kenneth M. Anderson  
University of Colorado, Boulder  
CSCI 4448/6448 — Lecture 2 — 08/28/2008

# Lecture Goals

---

- Introduce basic concepts, terminology, and notations for object-oriented analysis, design, and programming
  - A benefit of the OO approach is that the same concepts appear in all three stages of development
- Start with material presented in Appendix II of your textbook
  - Continue (in lecture 3) with additional material from previous versions of this class as well as from **Head First Java** by Sierra & Bates, © O'Reilly, 2003
- Will present examples and code throughout

# Big Picture View

---

- OO techniques view software systems as
  - **systems of communicating objects**
- Each object is an **instance** of a **class**
  - All objects of a class share similar **features**
    - **attributes**
    - **methods**
  - Classes can be **specialized** by subclasses
- Objects communicate by **sending messages**

# Welcome to Objectville

---

- What were the major concepts discussed in Appendix II of the textbook?
  - Unified Modeling Language (UML)
  - Class Diagrams
  - Inheritance
  - Polymorphism
  - Encapsulation

# UML

---

- UML stands for **Unified Modeling Language**
  - UML defines a standard set of **notations** for use in **modeling** object-oriented systems
  - Throughout the semester we will encounter UML in the form of
    - class diagrams
    - sequence/collaboration diagrams
    - state diagrams
    - activity diagrams, use case diagrams, and more

# Class Diagrams

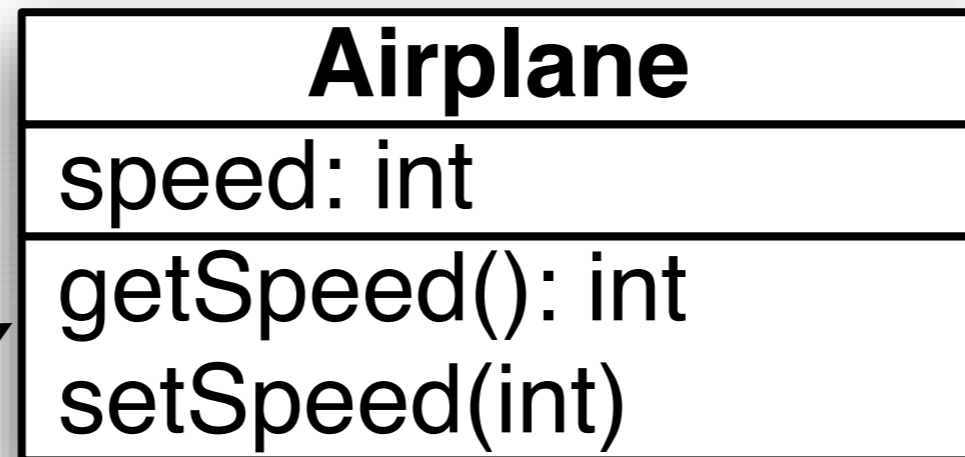
---

Class Name

All parts are optional  
except the class name

Attributes  
(member variables)

Methods



A class is represented as a rectangle

This rectangle says that there is a class called Airplane that could potentially have many instances, each with its own speed variable and methods to access it

# Translation to Code

---

- Class diagrams can be translated into code in a fairly straightforward manner
  - Define the class with the specified name
  - Define specified attributes (assume **private access**)
  - Define specified method skeletons (assume **public**)
- May have to deal with unspecified information
  - Types are optional in class diagrams
  - Class diagrams typically do not specify **constructors**
    - constructors are used to **initialize** an object

# Airplane in Java

---

```
1 public class Airplane {
2
3     private int speed;
4
5     public Airplane(int speed) {
6         this.speed = speed;
7     }
8
9     public int getSpeed() {
10        return speed;
11    }
12
13    public void setSpeed(int speed) {
14        this.speed = speed;
15    }
16
17 }
```



# Airplane in Python

---

```
1  class Airplane(object):
2
3      def __init__(self, speed):
4          self.speed = speed
5
6      def getSpeed(self):
7          return self.speed;
8
9      def setSpeed(self, speed):
10         self.speed = speed
```

# Airplane in Ruby

---

```
1  class Airplane
2
3      attr_accessor :speed
4
5      def initialize(speed)
6          @speed = speed
7      end
8
9  end
```

# Using these Classes?

---

- The materials for this lecture contains source code that shows how to use these classes
  - Demonstration
    - **Airplane.java, Airplane.py, Airplane.rb**
- Be sure to attempt to run these examples on your own
  - It will be good experience to learn how to run Java, Python, and Ruby programs on your personal machine or on a Lab machine (either ITS or CSEL)

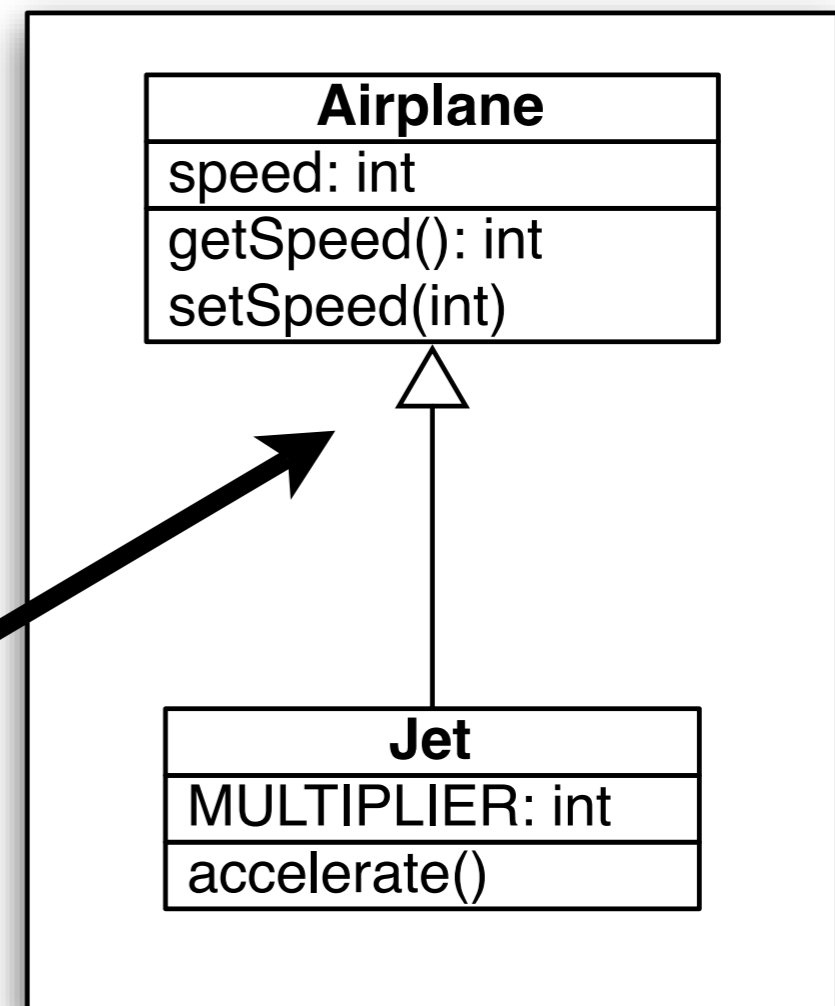
# Inheritance

---

- Inheritance refers to the ability of one class to **inherit behavior** from another class
  - and change that behavior if needed

Inheritance lets you build classes based on other classes and avoid duplicating and repeating code

Note: UML notation to indicate inheritance is a line between two classes with a triangle pointing at the **base class** or **superclass**



# Inheriting From Airplane (in Java)

---

```
1 public class Jet extends Airplane {
2
3     private static final int MULTIPLIER = 2;
4
5     public Jet(int id, int speed) {
6         super(id, speed);
7     }
8
9     public void setSpeed(int speed) {
10        super.setSpeed(speed * MULTIPLIER);
11    }
12
13    public void accelerate() {
14        super.setSpeed(getSpeed() * 2);
15    }
16
17 }
18
```

Note:

**extends** keyword indicates inheritance

**super()** and **super** keyword is used to refer to superclass

No need to define `getSpeed()` method; its inherited!

`setSpeed()` method **overrides** behavior of `setSpeed()` in Airplane

subclass **can define new behaviors**, such as `accelerate()`

# Inheritance in Python

---

```
1 class Jet(Airplane):
2
3     MULTIPLIER = 2
4
5     def __init__(self, id, speed):
6         super(Jet, self).__init__(id, speed)
7
8     def setSpeed(self, speed):
9         super(Jet, self).setSpeed(speed * Jet.MULTIPLIER)
10
11    def accelerate(self):
12        super(Jet, self).setSpeed(self.getSpeed() * 2);
13
```

# Inheritance in Ruby

---

```
1  class Jet < Airplane
2
3      @@MULTIPLIER = 2
4
5      def initialize(id, speed)
6          super(id, speed)
7      end
8
9      def speed=(speed)
10         super(speed * @@MULTIPLIER)
11     end
12
13     def accelerate()
14         @speed = @speed * 2
15     end
16
17 end
18
```

# Polymorphism: “Many Forms”

---

- From the textbook: “When one class inherits from another, then polymorphism allows a **subclass** to **stand in** for the **superclass**.”
- Implication: both of these are legal statements
  - Airplane plane = new Airplane()
  - Airplane plane = new Jet()
- Any code that uses the “plane” variable will treat it as an Airplane... this provides flexibility, since that code will run unchanged, **indeed it doesn't even need to be recompiled**, when new Airplane subclasses are created



# Encapsulation

---

- Encapsulation is
  - when you hide parts of your data from the rest of your application
  - and limit the ability for other parts of your code to access that data
- Encapsulation lets you protect information in your objects from being used incorrectly
  
- Closely Related Concept: Abstraction
  - What **features** does a class provide to its users?
  - What services can it perform?
  - Indeed, abstraction is the MOST IMPORTANT concern in OO A&D!!

# Encapsulation Example

---

- The “speed” instance variable is private in Airplane. That means that Jet doesn’t have direct access to it.
- Nor does any client of Airplane or Jet objects
- Imagine if we changed speed’s visibility to public
- The encapsulation of Jet’s setSpeed() method would be destroyed

```
1 Airplane
2
3 ...
4 public void setSpeed(int speed) {
5     this.speed = speed;
6 }
7 ...
8
9 Jet
10
11 ...
12 public void setSpeed(int speed) {
13     super.setSpeed(speed * MULTIPLIER);
14 }
15 ...
16
```

Demonstration

# Summary

---

- OO software is a **system of communicating objects**
- UML provides **standard notations** for **documenting the structure** of OO systems
- Classes define the **features** of objects, both their **data** and **behavior**
- Inheritance allows classes to **share behavior** and **avoid duplicating/repeating** code
- Polymorphism allows a **subclass** to **stand in** for its **superclass**
- Encapsulation occurs **when you hide parts of your code** from other parts, **thereby protecting it**

# Ken's Corner (I)

---

- Forgot to mention <http://slashdot.org/> on Tuesday
  - Developer-Oriented discussion forum with the motto:
    - News for Nerds. Stuff that Matters.
- Today's subject: Python Generators
  - Imagine you have been asked to create a function that produces a sequence of numbers, say the Fibonacci sequence
  - You might decide to take a parameter of the number of values to generate in the sequence and then return the values in a list

# Ken's Corner (II)

---

```
def fib_list(n):  
    results = []  
    a, b = 0, 1  
    for i in range(n):  
        results.append(b)  
        a, b = b, a+b  
    return results
```

`fib_list(5)` returns

```
[1, 1, 2, 3, 5]
```

# Ken's Corner (III)

---

- This approach works great as long as you don't pass in an unreasonable number for n
  - Why?
- To get around this problem, python added a feature called generators which are functions that contain at least one use of the "yield" keyword
  - When a generator encounters a yield, it saves all the values of its local variables and then returns the value specified by the yield keyword's expression
  - The next time the generator is called, execution begins just after the statement that contained the yield keyword, with all local variables restored to their previous value

# Ken's Corner (IV)

---

```
def fib():  
    a, b = 0, 1  
    while 1:  
        yield b  
        a, b = b, a+b
```

This produces the Fibonacci sequence one value at a time.

When you call `fib()`, you get back an iterator that can be used to loop over the values in the sequence.

You can either put the generator directly into a for loop or you can call the generator's `next()` function to get the next value of the sequence.

# Ken's Corner (V)

---

```
for i in fib():  
    print i
```

- The above will loop forever (or until some exception occurs).

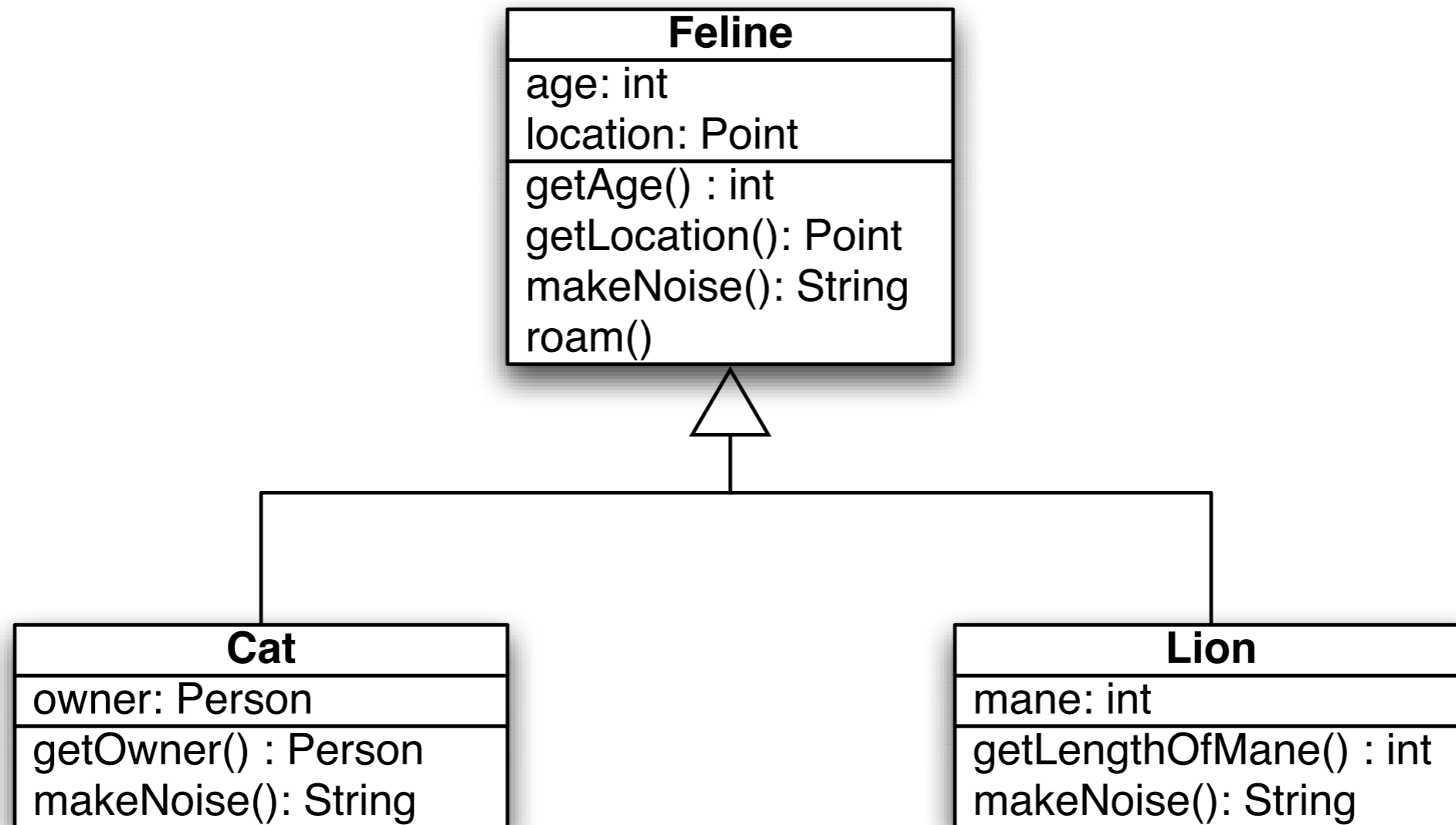
```
fib = fib()  
for i in range(100):  
    val = fib.next()  
    print val
```

- The above prints the first 100 values of the sequence
  - So generators are an excellent way of producing (possibly infinite) sequences in an efficient manner
  - They can be used in lots of different ways including when implementing state machines, parsers, and looping over large data sets



# Homework 1 (reminder: earns extra credit)

---



An optional homework assignment will appear on the website today that asks you to answer some questions about the above UML model. See the website for details.

# Coming Up Next

---

- Lecture 3: Object Fundamentals Continued
  - No reading assignment
  - Note: Lecture 3 **will repeat** some of the things mentioned in this lecture
- Lecture 4: Great Software
  - Read Chapter 1 of the OO A&D book