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
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
public class Airplane {

    private int speed;

    public Airplane(int speed) {
        this.speed = speed;
    }

    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        this.speed = speed;
    }

}
class Airplane(object):

    def __init__(self, speed):
        self.speed = speed

    def getSpeed(self):
        return self.speed;

    def setSpeed(self, speed):
        self.speed = speed
class Airplane

attr_accessor :speed

def initialize(speed)
  @speed = speed
end

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**

![UML Diagram](image)
public class Jet extends Airplane {
    private static final int MULTIPLIER = 2;
    public Jet(int id, int speed) {
        super(id, speed);
    }
    public void setSpeed(int speed) {
        super.setSpeed(speed * MULTIPLIER);
    }
    public void accelerate() {
        super.setSpeed(getSpeed() * 2);
    }
}

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

class Jet(Airplane):
    MULTIPLIER = 2

    def __init__(self, id, speed):
        super(Jet, self).__init__(id, speed)

    def setSpeed(self, speed):
        super(Jet, self).setSpeed(speed * Jet.MULTIPLIER)

    def accelerate(self):
        super(Jet, self).setSpeed(self.getSpeed() * 2);
Inheritance in Ruby

class Jet < Airplane
    @@MULTIPLIER = 2

    def initialize(id, speed)
        super(id, speed)
    end

    def speed=(speed)
        super(speed * @@MULTIPLIER)
    end

    def accelerate()
        @speed = @speed * 2
    end
end
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

```java
Airplane

... public void setSpeed(int speed) {
    this.speed = speed;
}
...

Jet

... public void setSpeed(int speed) {
    super.setSpeed(speed * MULTIPLIER);
}
... 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
Quick Exercise

• Develop a UML class diagram for the following two classes
  • A Person class that stores a person’s name, age, and favorite color
  • An Employee class that stores a person’s job title and salary
• Create the diagram such that the following Java code fragment would have a chance at running:
  • Employee e = new Employee( “Ken”, 41, “blue”, “Associate Professor”, 0);
  • System.out.println(“” + e.getAge());
  • Person p = ....
  • Employe e = new Employee(p, “prof”, 0);
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
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]
• This approach works great as long as you don’t pass in an unreasonable number for n
  
  • Why?

• Consider needing to process each element of this sequence, if you did this

• list = fib_list(some_huge_number)

• for i in list:
  
  • <do something>

• You need to essentially process the list twice:
  
  • first creating it and then looping over it

  • if it’s a really big list then you also can run into memory-related problems
Ken’s Corner (IV)

• 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
def fib(n):
    count = 0
    a, b = 0, 1
    while count < n:
        yield b
        count += 1
        a, b = b, a+b

This produces the first n numbers in the Fibonacci sequence, one value at a time. When you call fib(100), 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.
for i in fib(100):
    print(i)

• The above will print the first 100 values of the Fibonacci sequence
• Note that this solves the problem on slide 23
  • You can now loop over and process the sequence at the same time
    • You don’t have to create a potentially huge list of numbers and then loop over it
• So generators are an excellent way of producing (possibly infinite) sequences in an efficient manner
• Generators can be used in lots of different ways including when implementing state machines, parsers, and looping over large data sets
Ken’s Corner (VI)

• You don’t need to use the fib_list() style of code to create lists either
  • Instead, you can hand a generator to the “list constructor” and it will create a list from the generator
    • In this case, you do have to be aware of memory constraints
• results = list(fib(20))
• print(results)
  • would produce
    • [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
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