Lecture 7: Object-Oriented Concepts

Kenneth M. Anderson
Object-Oriented Analysis and Design
CSCI 6448 - Spring Semester, 2001

Goals for this Lecture

- Discuss History of Object Orientation
- Discuss Basic Concepts
  - Object
  - Class
  - Encapsulation
  - Inheritance
  - Composition
  - etc.

History of Object Orientation

- 1967 - Development of Simula
- 1970's SmallTalk
  - First "pure" object-oriented language
  - Java and C++ are not pure
- 1980's Graphical User Interfaces
  - 200 person-years of effort to develop the Apple Lisa; claim made that it would not have been possible without the inherent reusability of object-oriented code
- 1990's OO Databases, Distributed Systems, and OO Analysis and Design Methods
- 2000's Components and Software Architectures

OO Concepts

- Object-Orientation divides the world into objects that posses
  - (hidden) information
  - methods (or behavior)
  - a public interface
- This structure provides encapsulation
  - data and behavior are private; as long as the public interface remains unchanged, the data and behavior of an object can be freely modified

Benefits

- Two benefits from this approach are
  - **Understanding** of the system is easier as the semantic gap between the system and reality is small.
  - **Modifications** to the model tend to be local as they often result from an individual item, which is represented by a single object

Message Passing

- Objects communicate with each other via *message passing*
  - this prevents data duplication between objects
    - if you need information from another object, ask for it!
  - the types of messages that you can send to an object are determined by its public interface
  - the message passing system is separate from objects
    - this allows a caller to send a message to one object, but (unbeknownst to it) have its message received and processed by another object
    - this is known as *late binding*

50% Done!

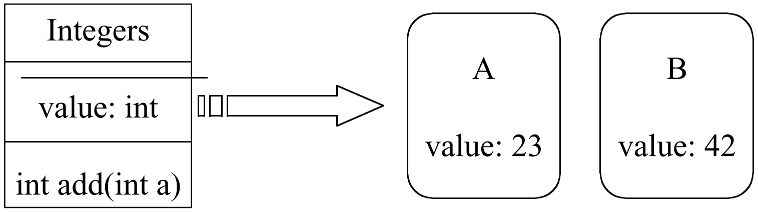
- You now know 50% of the idea behind object technology (in only two slides!)
  - Objects have public interfaces that hide data and behavior from the external world
  - Objects access this information via message passing
- The other 50% has to do with how we classify objects and relate them to one another

More on Objects

- Objects form the basic unit of OO A&D
  - They are instances organized into classes with common features
    - Attributes (previously called data)
      - these represent the object’s state or they capture associations with other objects
    - Operations/Methods (Behavior)
      - these are procedures or services that the object can perform
    - Invariants (new)
      - Rules that specify how the other features of the object are related or under what conditions the object is viable

Classes

- A class is a collection of objects which share common attributes and methods
  - A class can be regarded as a template for creating instances (e.g. objects)



Integers

value: int

int add(int a)

A

value: 23

B

value: 42

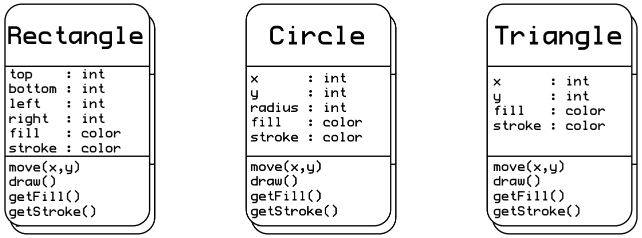
More on Classes

- A *type* is the specification of a class
  - A type represents ideas
    - Also known as the intension of a class
  - A type’s attributes and methods are known as its *features* or *responsibilities*
    - Attributes are a responsibility for knowing something; methods are a responsibility for doing something
- A class is the implementation of a type
  - It represents the collection of all objects that are instances of its type; known as the extension of a class

Object Relationships

- An object receives all of the attributes and methods of its class
  - this is known as *classification*
- It is possible for an object, however, to receive attributes and methods from more general *superclasses*
  - this is known as *inheritance*
- It is also possible for an object to consist of other objects (by pointing at them)
  - this is known as *composition*

Classification



Rectangle

```
top    : int
bottom : int
left   : int
right  : int
fill   : color
stroke : color
```

```
move(x,y)
draw()
getFill()
getStroke()
```

Circle

```
x      : int
y      : int
radius : int
fill   : color
stroke : color
```

```
move(x,y)
draw()
getFill()
getStroke()
```

Triangle

```
x      : int
y      : int
fill   : color
stroke : color
```

```
move(x,y)
draw()
getFill()
getStroke()
```

- Three Objects
  - Each with similar attributes and operations
  - They are instances of the rectangle, circle, and triangle classes

Discussion

- On the surface, there appears to be some duplication occurring
  - for instance, if we were to implement each of these operations, we would need three separate instances of the draw method, three separate instances of the move method, etc.
- We can use inheritance to address this situation
- Any suggestions?

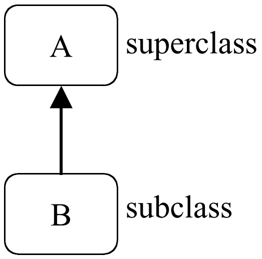
Inheritance

- Inheritance is a mechanism that enables *generalization* and *specialization*
  - generalization occurs when the common features of a set of classes is unified in a superclass
    - each member (potentially) retains its identity but now stores only those attributes and behavior specific to it
  - specialization occurs when a generic class is extended into a set of subclasses; each subclass shares the features of the generic class but has additional attributes and/or behaviors
  - thus, generalization/specialization are two sides of the same coin; it just depends on where you start

More on Inheritance

- Inheritance is a mechanism that lets
  - subclasses share attributes and methods with superclasses
    - therefore, if class A is a superclass of class B, and class A defines an attribute “age: int”, then B automatically has an attribute called age of type integer
    - furthermore, if class A has an operation “draw”, then class B automatically has an operation called draw;

Example Illustrated

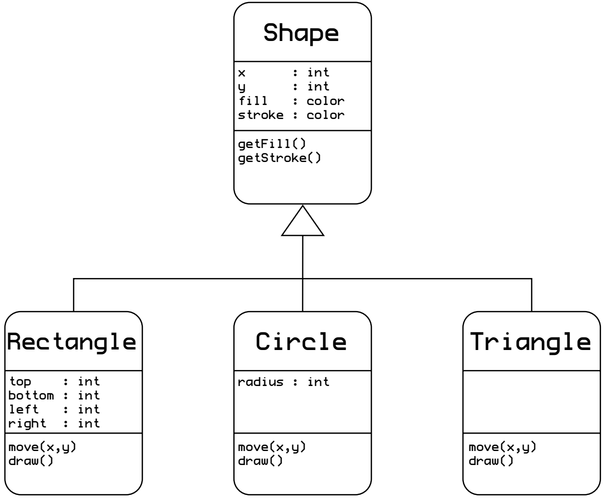


If A defines an attribute called age, then we can set a value for that attribute in B, because B inherits that attribute from A

Thus, b.age = 10 is perfectly legal, even if B s class definition says nothing about an age attribute

In the same manner, if A defines a method called draw() but B does not, it is still legal to say b.draw() because when we pass the draw message to B, it will look for a method called draw first in B, and then in A, thus b.draw() will result in the draw method defined by A to execute.

Earlier Example Revisited



Substitutability

- One benefit of inheritance is the notion of substitutability
  - since a subclass supports all of the methods that its superclass supports, a subclass can “stand in” or “substitute” for the superclass
  - Thus if I have a class called Shape (of which Rectangle, Circle, and Triangle are subclasses) then I can say things like
    - myVariable: Shape
    - myVariable := new Circle()
    - myVariable.getFill()

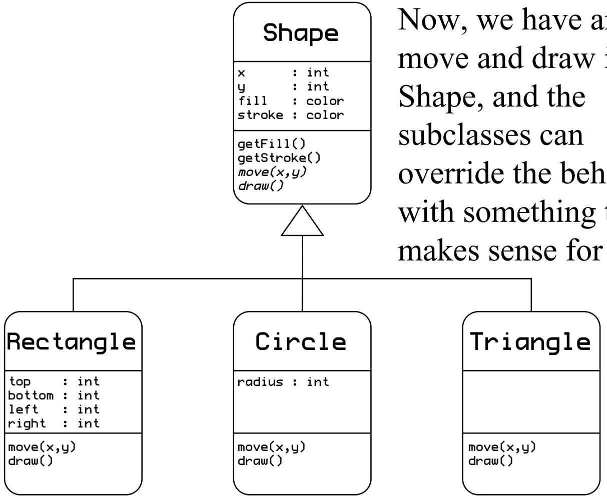
Overriding

- A benefit of inheritance is that subclasses can override the behavior of their superclasses
  - that is, they can change the behavior of the inherited methods
  - this is a powerful feature, but it is at odds with substitutability
    - the greater the change in behavior, the less the subclass is able to “stand in” for its superclass

Why do we need overriding?

- Consider our shape example, currently we do not have the routines “draw” and “move” defined in the Shape class
  - because each of these routines need to do something different based on their shape
  - but superclasses are supposed to contain “common features” of its superclass; so here we have three subclasses each with a draw and move routine that does not appear in the superclass
  - to fix this; we can add move and draw to Shape, but make them null-ops, also known as “abstract”

Example, continued



```
Shape
x      : int
y      : int
fill   : color
stroke : color
getFill()
getStroke()
move(x,y)
draw()
```

```
Rectangle
top    : int
bottom : int
left   : int
right  : int
move(x,y)
draw()
```

```
Circle
radius : int
move(x,y)
draw()
```

```
Triangle
move(x,y)
draw()
```

Now, we have an move and draw in Shape, and the subclasses can override the behavior with something that makes sense for them.

Polymorphism

- The concept of polymorphism takes advantage of this new configuration to provide even more power
  - simply put, polymorphism routes calls to a method of a superclass to its subclasses if the subclass has overridden the behavior of the superclass
  - we can now create generic algorithms with respect to the Shape class, that is, we can create an array of Shape objects (rectangles, circles, etc.), call move on each instance, and the correct move routine will be called based on that instance's type

Polymorphism Example

```
ShapeArray a = {new circle(), new
 rectangle(), new triangle()}
Shape x
for i = 1 to a.size() {
  x = a(i);
  x.move(0,x.y)
}
```

- first circle.move() is called, then rectangle.move(), then triangle.move() even though the type of x is “Shape”

Composition

- Objects can also participate in composition relationships
- In composition, one object encapsulates another and uses the internal object to help meet its own obligations
- For instance, a Window object may contain a Rectangle object (as an attribute); it can use the Rectangle object to help keep track of its coordinates, calculate its area, determine if its overlapping some other Rectangle, etc.

Aggregation

- Composition is sometimes referred to as aggregation
  - Aggregation is somewhat different from the sense of composition used in the previous slide
  - The classic example for aggregation is a Car object; it is composed of a number of wheel objects, door objects, instrument objects, an engine object, etc.
- In practice, composition relationships can model a variety of associations; we will learn more about this later in the semester

Summary

- OO divides the world into objects
  - each object has attributes (state), methods (behavior), an interface, and (sometimes) constraints
  - the interface hides the details of the attributes and the methods (encapsulation)
  - objects communicate by sending each other messages
  - objects can be arranged in various ways including inheritance and composition
  - inheritance enables overriding and polymorphism