CS 461

Programming Language Concepts

Gang Tan
Computer Science and Engineering
Penn State University

Overview of OO Programming

- OO program: collection of objects which communicate by message passing
- Generally, only one object is executing at a time

◆Programming methodology
- organize concepts into objects and classes
- build extensible systems through subtyping (subclassing) and inheritance

Object-Orientation

◆Language concepts
- Dynamic lookup
- Method overloading
- Subtyping allows extensions of concepts
- Inheritance allows reuse of implementation
- Method overriding

Dynamic Lookup

◆ AKA, dynamic dispatch
◆ In dynamic lookup,
object.method(arguments)
which method is invoked is determined dynamically: depends on the real runtime class of the object and arguments
- The real class might be different from the static class (compiler view) of the object

◆ In static lookup,
function (arguments)
meaning of function is statically determined

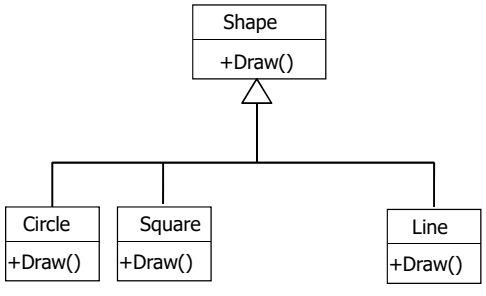
Fundamental difference between abstract data types and objects

Example

◆Add two numbers: x.add (y)
different add if x and y are integer, float, complex

◆Conventional programming: add (x, y)
function add has fixed meaning

UML Class Diagram for Shapes



Shapes in Java

```
abstract class Shape {
    abstract void Draw ();
}

class Circle extends Shape {
    void Draw () {…}
}

class Square extends Shape {
    void Draw () {…}
}

class Line extends Shape {
    void Draw () {…}
}
```

7

Drawing Shapes

```
void drawShapes(Shape s[]) {
        for (int i=0; i<s.length; i++) {
            s[i].Draw();
        }
}
…
Shape s[] =
  {new Circle(), new Square(), new Line()};
drawShapes(s);
```

8

Shapes in Python, Part I

```
class Shape:
    def Draw(self):
        raise Exception('calling an abstract method')

class Circle(Shape):
    def Draw(self):
        print 'drawing a circle'

class Square(Shape):
    def Draw(self):
        print 'drawing a square'

class Line(Shape):
    def Draw(self):
        print 'drawing a line'
```

9

Shapes in Python, Part II

```
lst = [Circle(), Square(), Line()]
for s in lst:
    s.Draw()
```

10

Dynamic Lookup in C++

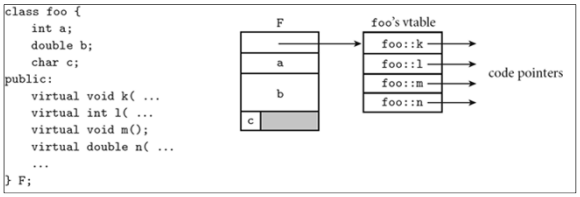
- By default, invocation of a member method uses static lookup
- But if a member method is declared with virtual, it uses dynamic lookup

11

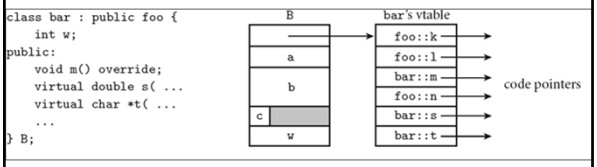
Implementation of Dynamic Method Binding

- They are implemented by creating a dispatch table (vtable) for the class and putting a pointer to that table in the data of the object
- Objects of a derived class have a different dispatch table
  - In the dispatch table, functions defined in the parent come first, though some of the pointers point to overridden versions
  - You could put the whole dispatch table in the object itself, saving a little time, but potentially wasting a LOT of space

Implementation of Dynamic Method Binding



Implementation of Dynamic Method Binding



Overloading

- Method Overloading
  - Creation of several methods with the same name but with parameters of different numbers and types
  - Compile time: uses the number of arguments and their types to decide which one to invoke
- General overloading
  - Overload the same symbol with multiple meanings
  - Examples of general overloading:
    - +, -, *, / can be float or int
    - + can be float or int addition or string concatenation in Java

Dynamic lookup vs Method Overloading

- Method overloading
  - Resolved at compile-time about which method is used
    - Used by statically typed languages
    - Python does not allow method overloading
- Dynamic lookup
  - Resolved during runtime

16

Example 1

```
class A {
    protected int x;
    A (int x) {this.x = x;}
    public int mult(int y) {return x*y;}
    public float mult(float y) {return x*y;}
}

class B extends A {
    B(int x) {super(x);}
    public int mult(int y) {
        System.out.println("In B");return x*y;}
    public float mult(float y) {
        System.out.println("In B");return x*y;}
}
```

17

Example 1

```
A a = new A(2);
System.out.println(a.mult(3));
System.out.println(a.mult((float)3.14));

A ab = new B(2);
System.out.println(ab.mult(3));
System.out.println(ab.mult((float)3.14));
```

6

6.28

In B; 6

In B; 6.28

18

Example 2

```
class A {
    public void display(A a) {
          System.out.println("In A.display(A)");}
    public void display(B b) {
          System.out.println("In A.display(B)");}
}

class B extends A {
    public void display(A a) {
          System.out.println("In B.display(A)");}
    public void display(B b) {
          System.out.println("In B.display(B)");}
}

A ab = new B();
ab.display(ab);
```

In B.display(A)

19

Example 3

```
class A {
    A () {}

    public void display(A a) {System.out.println("In A.display(A)");}
    // public void display(B b) {System.out.println("In A.display(B)");}
}

class B extends A {
    B() {}

    public void display(A a) {System.out.println("In B.display(A)");}
    public void display(B b) {System.out.println("In B.display(B)");}
}
```

20

Example 3

```
A a = new A();
B b =new B();
A ab = new B();

ab.display(b);
ab.display(a);
b.display(ab);
b.display(b);
```

In B.display(A)

In B.display(A)

In B.display(A)

In B.display(B)

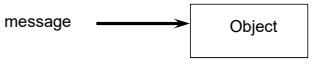
21

Subtyping, inheritance and method overriding

22

Encapsulation (Abstraction)

- External view: User of a concept has "abstract" view
  - Interface
- Internal view: Builder of a concept has detailed view
  - Implementation
- Encapsulation separates these two views
  - Implementation code: operate on representation
  - Client code: operate by applying a fixed set of operations provided by implementer of abstraction



Object Interfaces

- Interface
  - The messages understood by an object
- Example: point
  - x-coord : returns x-coordinate of a point
  - y-coord : returns y-coordinate of a point
  - move : method for changing location
- The interface of an object is its *type*.

Subtyping: Relation Between Two Interfaces (External Views)

- If interface A contains all of interface B, then A is a subtype of B

Point	Colored_point
x-coord	x-coord
y-coord	y-coord
move	color
	move
	change_color

- Colored_point interface contains Point
  - Colored_point is a subtype of Point
- If A is a subtype of B, then A objects can also be used as B objects
  - Liskov substitution principle

Inheritance: Relation Between Implementations (Interval Views)

- Relation between two implementations
- New objects may be defined by reusing implementations of other objects
- New objects can also override the implementation of methods of the inherited object
  - Method overriding

Method overriding

- Not the same as method overloading
- Java: Only override the method with the same type

```
class A {
    public void display(A a) { …}
    public void display(B b) { … }
}
class B extends A {
   public void display(A a) {…}
   // A.display (B b) available in B
}
```

27

Single Inheritance

- The class hierarchy forms a tree
- A class has exactly one parent class
  - except for the root class
- Rooted in a most general class: Object
- Single inheritance languages: Smalltalk, Java

Multiple Inheritance

- Allows a class to be a subclass of zero, one, or more classes.
- Class hierarchy is a directed acyclic graph (dag)
- Pros: facilitates code reuse
- Cons: more complicated semantics
  - E.g., if two parent classes have the same method, which one's implementation to inherit?
- Example: Python
  - class DerivedClassName(Base1, Base2, Base3):
    …
  - Impose a search order to look for attributes
    - depth-first, left-to-right
    - E.g., if both Base1 and Base 2 have attribute a, then Base1's a is inherited

Python Example for Multiple Inheritance, part I

```
class circle:
    def __init__(self, radius):
        self.radius = radius

    def __repr__(self):
        return "I am a circle"

    def area(self):
        return 3.14 * self.radius * self.radius
```

30

Python Example for Multiple Inheritance, part II

```
class coloredShape:
    def __init__(self, color):
        self.color = color

    def __repr__(self):
        return "My color is " + self.color

    def getColor(self):
        return self.color

    def changeColor(self, color):
        self.color = color
```

31

Python Example for Multiple Inheritance, part III

```
class coloredCircle(coloredShape, circle):
    def __init__(self, radius, color):
        circle.__init__(self,radius)
        coloredShape.__init__(self,color)

cc = coloredCircle(3, "Blue")
print cc.area()
print cc.getColor()
cc.changeColor("Green")
print cc.getColor()

print cc
```

32