CSE 3302
Programming Languages
Lecture 7: Object-Oriented Programming

(based on the slides by Weimin He & Chengkai Li)
Leonidas Fegaras
University of Texas at Arlington
Introduction of OO programming using Java

• Why Java?
  – Simple
  – Platform Independent
  – Safe
  – Multi-Threaded
  – Uses Garbage Collection
Objects, Classes, and Methods

• **Object**
  Something that occupies memory and has a state

• **Class**
  Pattern (type) of object: the object description

• **Method**
  Function or procedure to access object state
Constructor Overloading

public class Point
{
    ... 
}

public class Rectangle
{
    private int width = 0;
    private int height = 0;
    private Point origin;

    public Rectangle()
    {
        origin = new Point(0, 0);
    }

    public Rectangle(Point p, int w, int h)
    {
        origin = p;
        width = w;
        height = h;
    }

    ...
}
Access Control

- **Public:** no restrictions
- **Private:** inaccessible from code outside the class definition
- **Protected:** can be accessed by classes that extend this class
  - ... only through a reference of that class/subclass
- **Package:** protected members can be accessed by all classes within the same package
Example of Access Control

Base.java

```java
package p1;
public class Base {
    private int u;
    protected int v;
    public int w;
    private void g () {
        System.out.println(v);
    }
    protected void f () {
        System.out.println(u);
        g();
    }
}
```

Derived.java

```java
package p2;
import p1.Base;
public class Derived extends Base {
    void h () {
        f();
    }
    void k () {
        g();
    }
}
```

Test.java

```java
package p2;
import p1.Base;
public class Test {
    public static void main(String[] args) {
        Base b = new Base();
        b.u = 1;
        // Compilation error
        b.v = 2;
        // Compilation error
        b.w = 3;
        // OK
        b.g();
        // Compilation error
        b.f();
        // Compilation error
        Derived d = new Derived();
        d.h();
        // OK
        d.k();
        // Compilation error
    }
}
```
Inheritance

```
public class B extends A {
    ...
}
```

```
public class A {
    ...
}
```
class Object
{
    ...
    public boolean equals ( Object obj ) { /* ... */ }
    public String toString() { /* ... */ }
    ...
}
Method equals ()

• == identical references to objects
  String s = “Hello”;
  String t = new String(“Hello”);
  if ( s == t ) System.out.println( “the same”);
    // condition is false and prints nothing!
  if ( s.equals(t) ) System.out.println( “the same”);
• equals: equality of objects
  – Already implemented in class Object
  – Need to define for your class :

  Point p1 = new Point (10, 10);
  Point p2 = new Point (5, 5);
  Rectangle rec1 = new Rectangle (p1, 5,10);
  Rectangle rec2 = new Rectnagle (p2, 5, 10);
  if ( rec1 == rec2 ) System.out.println(“the same”);
    // condition is false and prints nothing!
  if ( rec1.equals(rec2) ) System.out.println(“the same”);
public class Rectangle
{
    ...
    public boolean equals ( Object obj )
    {
        Rectangle rec = (Rectangle) obj;
        return (w == rec.width()) && (h == rec.height());
    }
}
Rectangle rec1 = new Rectangle (p1, 5, 10);
Rectangle rec2 = new Rectangle (p2, 5, 10);
if ( rec1.equals(rec2) ) System.out.println("the same!");
// condition is true and prints “the same”!
Method toString() 

Rectangle rec = new Rectangle (p1, 5, 10);
System.out.println( rec );

// prints something like Rectangle@73d6at
(class name +@+ hash code of object)
public class Rectangle
{
    ...

    public String toString()
    {
        return "width = " + w + " , height = " + h;
    }

}

Rectangle rec = new Rectangle (p1, 5, 10);
System.out.println( rec );

Output:
width = 5 , height = 10   (instead of Rectangle@73d6at)
Inheritance

public class Circle
{
    private Point center;
    private double radius;

    public Circle(Point c, double r)
    {
        center = c;
        radius = r;
    }

    public double area()
    {
        return Math.PI * radius * radius;
    }
}

public class Rectangle
{
    private Point center;
    private double width;
    private double height;

    public Rectangle(Point c, double w, double h)
    {
        center = c;
        width = w;
        height = h;
    }

    public double area()
    {
        return width * height;
    }
}
public abstract class Shape
{
  private Point center;

  public Shape(Point p)
  {
    center = p;
  }

  public abstract double area();
}
Inheritance (Cont’d)

```java
public class Circle extends Shape {
    public Circle( Point c, double r) {
        super(c);
        radius = r;
    }

    public double area() {
        return Math.PI * radius* radius;
    }
}
```

```java
public class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle( Point c, double w, double h) {
        super(c);
        width = w;
        height = h;
    }

    public double area() {
        return width * height;
    }
}
```
Casting

• **Upcasting:**
  Assign a subclass reference to a superclass variable

• **Downcasting:**
  Assign Superclass Reference to subclass variable
Example of Casting

```java
public class Queue {
    // constructors and instance variables
    // omitted
    public void enqueue( int x) { … }
    public void dequeue() { … }
    public int front() { … }
    public boolean empty() { … }
}

public class Deque extends Queue {
    …
    public void addFront( int x ) { … }
    public void deleteRear() { … }
}

Queue q1, q2;
Deque d;
d = new Deque();
q1 = d;
q1.deleteRear();
// Compilation error
d = q1;
// Compilation error
d = (Deque) q1;
d.deleteRear();
q2 = new Queue();
d = (Deque) q2;
// Runtime error
d.deleteRear();
```
Dynamic Binding

• Dynamic Binding
  – A mechanism by which the runtime system (JVM) selects the appropriate method at runtime, based on the class of the object
    • when the compiler can't determine which method implementation to use in advance
  – The process of binding a call to a particular method
    • This is performed dynamically at run-time
Dynamic Binding Example

class A
{ void p() { System.out.println("A.p"); } 
  void q() { System.out.println("A.q"); } 
  void f() { p(); q(); } 
}
class B extends A
{ void p() { System.out.println("B.p"); } 
  void q() { System.out.println("B.q"); super.q(); } 
}
public class Test 
{ public static void main(String[] args) 
  { A a = new A(); 
    a.f(); 
    a = new B(); 
    a.f(); } 
} 

Print:
A.p
A.q
B.p
B.q
A.q
Dynamic Binding Example (Cont’d)

class A {
    public void q(){
        System.out.println("A.q()");
    }
}
class B extends A {
    public void f(){
        System.out.println("B.f()");
        super.q();
    }
}
class C extends B {
    public void q() {
        System.out.println("C.q()");
    }
}
public class D extends C {
    public static void main(String[] args){
        D d = new D();
        d.f();
    }
}
Abstract Class

- Abstract classes
  - Cannot be instantiated
  - Incomplete: subclasses fill-in "missing pieces"

- To make a class abstract
  - public abstract class Shape {...}
  - Contains one or more abstract methods
    - No implementation
    - E.g., public abstract void draw();

- Subclasses:
  fill in "missing pieces" (i.e., overriding the abstract methods)
  - E.g., Circle, Triangle, Rectangle extends Shape
    - Each must implement draw (it is concrete)
Example of Abstract Class

public class Base
{
    public String m1()
    {
        return "Base.m1";
    }
}

public abstract class Derived extends Base
{
    public abstract String m2();
}

public class Derived2 extends Derived
{
    public String m2()
    {
        return "Derived2.m2";
    }
}
• No method implementations

• Java doesn’t allow multiple inheritance:
  – E.g., … C extends A, B …

• Instead, use Interface
  – E.g., … C implements I₁, I₂ …

• One class may implement multiple interfaces
  – Must implement all functions in those interfaces if class is concrete
public class Base
{
    public String m1()
    {
        return "Base.m1";
    }
}

interface Interface1 { String m2(); }

interface Interface2 { String m3(); }

public class Derived extends Base implements Interface1, Interface2
{
    public String m2()
    {
        return "Derived.m2";
    }

    public String m3()
    {
        return "Derived.m3";
    }
}
interface Interface1 { String m( String s ); }

interface Interface2 { String m( int i ); }

public class Derived implements Interface1, Interface2 {
    public String m(String s) {
        return "Derived.Interface1.m";
    }
    public String m(int i) {
        return "Derived.Interface2.m";
    }
}
interface Interface1 { String m( String s ); }

interface Interface2 { String m( String s ); }

public class Derived implements Interface1, Interface2 {
    public String m(String s) {
        return "Derived.Interface1&Interface2.m";
    }
}

Interface Example (Cont’d)
Interface (Java) vs. Multiple Inheritance (C++)

abstract class A {
    abstract public void f();
}

class B extends A {
    public void f() {
        System.out.println("B.f()");
    }
}

class C extends A {
    public void f() {
        System.out.println("C.f()");
    }
}

public class D extends B, C {
    public static void main(String[] args) {
        D d = new D();
        d.f();
    }
}
Interfaces vs. Abstract Classes

- A class may implement several interfaces
- An interface cannot provide any code at all
- Static final constants only

- A class may extend only one abstract class
- An abstract class can provide partial code
- Both instance and static constants are possible
public class Base
{
    public String m1()
    {
        return "Base.m1";
    }
}

interface Interface1
{
    String m1()
    String m2();
}

public abstract class Derived extends Base implements Interface1
{
    public String m2()
    {
        return "Derived.m2";
    }
}

public class Derived2 extends Derived
{
    public String m1()
    {
        return "Derived2.m1";
    }
}

Derived2 derived2 = new Derived2();
Base base = derived2;

tmp = derived2.m1();
    // tmp is "Derived2.m1"

tmp = base.m1();
    // tmp is "Base.m1"

tmp = derived2.m2();
    // tmp is "Derived.m2"
String Processing

• Class java.lang.String
• Class java.lang.StringBuffer
• Class java.util.StringTokenizer
Important Methods in Class String

• Method `length`
  – Determine String length
    • Like arrays, Strings always “know” their size
    • Unlike arrays, Strings do not have length instance variable

• Method `charAt`
  – Get the character at specific location in String

• Method `getChars`
  – Get an entire set of characters in String

• Method `startsWith`
  – Tests if this string starts with the specified prefix

• Method `split`
  – Splits this string around matches of the given regular expression
Important Methods in Class String

• Comparing String objects
  – Method `equals`
  – Method `equalsIgnoreCase`
  – Method `compareTo`

• Search for characters in String
  – Method `indexOf`
  – Method `lastIndexOf`

• Create Strings from other Strings
  – Method `substring`

• Concatenate two String objects Method `concat`
  – Method `concat`

• String provides static class methods
  – Method `valueOf`
    • Returns the String representation of object, data, etc.
Class StringBuffer

- Class StringBuffer
  - When String object is created, its contents cannot change
  - Used for creating and manipulating dynamic string data
    - i.e., modifiable Strings
  - Can store characters based on capacity
    - Capacity expands dynamically to handle additional characters
  - Uses operators + and += for String concatenation
Important Methods in StringBuffer

• Method charAt
  – Returns StringBuffer character at specified index

• Method setCharAt
  – Sets StringBuffer character at specified index

• Method getChars
  – Returns the character array from StringBuffer

• Method append
  – Allows data values to be added to StringBuffer

• Method reverse
  – Reverses StringBuffer content

• Method insert
  – Allows data-type values to be inserted into StringBuffer

• Methods delete and deleteCharAt
  – Allows characters to be removed from StringBuffer
Important Methods in Class StringBuffer

• Method `length`
  – Returns `StringBuffer` length

• Method `capacity`
  – Returns `StringBuffer` capacity

• Method `setLength`
  – Increases or decreases `StringBuffer` length

• Method `ensureCapacity`
  – Sets the `StringBuffer` capacity
  – Guarantees that `StringBuffer` has minimum capacity
Class StringTokenizer

- **Tokenizer**
  - Partition `String` into individual substrings
  - Use *delimiter*
  - Java offers `java.util.StringTokenizer`
Class StringTokenizer

- Constructor `StringTokenizer(String str)`
  - Constructs a string tokenizer for the specified string
- Constructor `StringTokenizer(String str, String delim)`
  - Constructs a string tokenizer for the specified string
- Method `hasMoreTokens()`
  - Tests if there are more tokens available from this tokenizer's string
- Method `nextToken()`
  - Returns the next token from this string tokenizer
Why Study Smalltalk

- Purest OO language, encourages OO programming
- Can inspect and change objects and the runtime system itself at run time
- Pioneer in many other areas
  - Graphical user interface (Window, Menu, Mouse)
  - Personal workstation
  - Pushed OO into success
    - I invented the term Object-Oriented, and I can tell you I did not have C++ in mind. -- Alan Kay
- Has an active community
- Could have deserved more popularity
Key Features

- Very simple syntax
- Everything is an object
  3, true, nil
  Class is object
  no control: if, loop are objects
- Dynamically typed:
  - A variable has no type
  - The class hierarchy is the type system
- The language is tightly combined with its interactive runtime system
  - The runtime system written in the language itself
  - Can change the system on-the-fly
  - Can debug system state (image), object, class hierarchy
- All data are private (protected), all methods are public
- No manual memory management, no pointers
History

• 1967: Inspired by Simula67, the first OO language
• 1971: Started by Dynabook project (Alan Kay)
  – Hardware: GUI, pointing device, external storage, etc.
    • Later led to Alto (“laptop”)
  – Software: for Children
    • Became Smalltalk
  – Education

Is Dynabook realized? Kay doesn’t think so:
– Squeak
– One Laptop Per Child project (Nicholas Negroponte, Alan Kay, …)
  http://www.xogiving.org/
History (cont.)

• 1973: Xerox Alto Computer
  – First Smalltalk environment
  – The ALTO Computer 1974

• 1972: Smalltalk-72

• 1976: Smalltalk-76

• 1980: Smalltalk-80

• Major contributors of Smalltalk:
  Alan Kay (2003 Turing Award), Adele Goldberg, Daniel Ingalls, et. Al.
Smalltalk Today

- **Squeak**
  - **Seaside**: a dynamic web application development framework
  - **Croquet**: an open source platform for collaborative 3D multi-user online applications
  - **Sophie**: a digital media assembly tool to create multimedia documents
  - **Scratch**: a toolkit for children to write games, animated stories

- **Cincom VisualWorks, IBM VisualAge, GNU Smalltalk, Dolphin, ….**
Object Hierarchy

- Object
  - UndefinedObject
  - Boolean
    - True
    - False
  - Magnitude
    - Char
    - Number
      - Fraction
      - Integer
      - Float
  - Collection
    - Set
    - ...
Syntax

• Smalltalk is really “small”
  – Only 6 keywords (pseudo variables)
  – Class, object, variable, method names are self explanatory
  – Only syntax for calling method (messages) and defining method.
    • No syntax for control structure
    • No syntax for creating class
Expressions

• Literals
• Pseudo Variables
• Variables
• Assignments
• Blocks
• Messages
Literals

- Number: 3 3.5
- Character: $a$
- String: ‘ ’ (‘Hel’, ’lo!’ and ‘Hello!’ are two objects)
- Symbol: # (#foo and #foo are the same object)
- Compile-time (literal) array: #(1 $a 1+2)
- Run-time (dynamic) array: {1. $a. 1+2}
- Comment: “This is a comment.”
Pseudo Variables

- **true**: singleton instance of True
- **false**: singleton instance of False
- **nil**: singleton instance of UndefinedObject
- **self**: the object itself
- **super**: the object itself (but using the selectors defined for the superclass)
- **thisContext**: activation of method (inspect the state of system)
Variables

- **Instance variables**

- **Local variables (method, blocks)**

  ```
  | sampleCell width height n |
  ```

- **Arguments (method argument, block argument)**
  - **method argument:**
    ```
    aGame toggleNeighboursOfCellAt: i at: j
    ```
  - **block argument:**
    ```
    [ :i :j | self newCellAt: i at: j ]
    ```

- **Shared Variables:**
  - **Global variables, e.g.,** Transcript
  - **Class variables, e.g.,** Epsilon in Float
Conventions

- Class name, class variable, global variable:
  (Capital letter for the first character of every word)
  Table
  HashTable

- Local variables, arguments, instance variable:
  (Capital letter for the first character of every word, except the first word)
  sampleCell

- Object (instance of a class, especially arguments)
  aTable
  aHashTable
Assignments

- bounds := 0@0 corner: 16@16
  or
- bounds <- 0@0 corner: 16@16

- Assignment returns a value, which is the object to the left of :=
Defining a Method

selector (method name)
  | local variable |
statement (expression).
statement(expression).
^ return-value

( . is used to end a statement)
(^ returns value from a method)
Methods and Messages

- Method Name: **Selector**
- Method Invocation: **Message**
  - Unary selector
    
    \[
    \text{3 factorial}
    \]
  
  object selector message

- Keyword selector
  
  \[
  \text{3 raiseTo: 2}
  \]

object selector (raiseTo:) message

‘Programming Language’ indexOf: $a startingAt: 3
Keyword Selector: more readable

• table insert: anItem at: anIndex

  table insert: 3 at: 5

vs.

• table.insert(anItem, anIndex)

  table.insert(3,5)
Binary selector

- \[ 2 + 3 \]
- \[ 2 + 3 + 4 \]
- \[ aTable / 3 \] (what it means depends on the class)
- \[ 1 + 2 \times 3 \] (\( \times \) does not have higher precedence than \(-\), because they are messages that can be sent to any object. No mathematical meaning is assumed.)

- Examples:
  \[ 3 / 5 \]
  \[ (1 / 3) + (1 / 2) \]
Binary selector

- + - * /
- = (equality) ~= >= <= > <
- == (identity, the two objects are the same object), ~
- & | Boolean
- , (string concatenation)

‘Hel’,’lo’ = ‘Hello’
‘Hel’,’lo’ == ‘Hello’
#Hello == #Hello

• Assignment := is not a method
Expression

• **Associativity for unary selector**: left to right
  
  3 factorial isPrime

• **Associativity for binary selector**: left to right
  
  1+2/4

• **Precedence rules:**
  
  Unary selector, then Binary selector, then Keyword selector
  
  2 raisedTo: 1 + 3 factorial

• ( ) for changing the order of evaluation

• “-object” was not there originally. So “3 - - 4" generated syntax errors in previous versions.
Message Cascading

• i.e., Sequence Operator

  Transcript cr.
  Transcript show: 'hello world'.
  Transcript cr

  Transcript cr; show: 'hello world'; cr
Block

A block is an anonymous function
Evaluate a block: value

*The evaluation result is the object from the last statement.*

[ 1+2 ] value
[ 1+2. ‘abc’, ‘def’] value
[ 1+2. SBEGame new] value
Block Parameters

- \[ [ :x :y \mid x+y ] \text{ value: 2} \text{ value: 3} \]
- \[ [ :x :y \mid \\
  \mid z \mid \\
  z := x + y. \\
  z := z \times z. \\
  z \]
  \text{ value: 2} \text{ value: 3} \]
Block Closure

- Block can access variables declared in enclosing scope.

| x |

x := 1.

[ :y | x + y ] value: 2.

A Block is an Object!

\[ z := [ :x :y \mid x+y ]. \]

\[ z \text{ value:2 value:3} \]
“Control Structures” by Messages

• Conditions: Messages to Boolean objects, with blocks as arguments

class True (subclass of Boolean, False is similar)

Selectors:

  ifTrue: alternativeBlock
    ^ alternativeBlock value
  ifFalse: alternativeBlock
    ^ nil

• Example
  – (a < b) ifTrue: [max:=b] ifFalse: [max:=a]
• While Loops: blocks as message receivers
• Example
  – n := 1.
    [ n < 10 ] whileTrue: [ n := n*2 ]
• **Counting Loops:** blocks as parameters

• **Example**

  - ```
    n := 1.
    10 timesRepeat: [ n := n*2 ]
  ```

  - ```
    n := 1.
    1 to: 10 do: [ n := n*2 ]
  ```

  - ```
    n := 0.
    1 to: 10 by: 2 do: [ i | n := n + i ]
  ```

  - ```
    n := 0.
    10 to: 1 by: -2 do: [ i | n := n + i ]
  ```