UCLA PIC 20A
Java Programming

- **Instructor:** Ivo Dinov,
  Asst. Prof. In Statistics, Neurology and Program in Computing
- **Teaching Assistant:** Yon Seo Kim, PIC

University of California, Los Angeles, Summer 2002

http://www.stat.ucla.edu/~dinov/

Chapter 5 – Classes & Inheritance

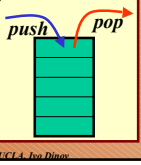
- Creating Classes
- Managing Inheritance
- Nested Classes

Creating Classes

```
public class Stack {                                   <- Class Declaration
    private Vector items;                              <- Variable
    public Stack() { items =new Vector(10); }          <- Constructor
    public Object push (Object item) {
        items.addElement(item); return item; }
    public synchronized Object pop()  {
        int len =items.size();
        Object obj =null;
        if (len ==0) throw new EmptyStackException();
        obj =items.elementAt(len -1);
        items.removeElementAt(len -1);
        return obj;
    }
    public boolean isEmpty()  {
        if (items.size()==0)  return true;
        else   return false;
} }
```

Class Body

Methods



Class Definition = Class Declaration + Class body

public Class // is publicly accessible.

abstract Class // cannot be instantiated.

final Class // cannot be subclassed.

class *NameOfClass* **extends** *Super*
implements *Interfaces* // *ClassBody*
{ // Can extend 0 or 1 SuperClass
// **Can implement many** *Interfaces*
}

Declaring Member Variables

accessLevel (= public, protected, package, and private) Lets you control what other classes have access to a member variable

static - Declares a class variable rather than an instance variable.

final - INDICATES that the value of this member cannot change.

transient - Marks member variables that should not be serialized. This component is used in object serialization (Interface Serializable).

volatile - Prevents the compiler from performing certain optimizations on a member.

type - Like other variables, a member variable must have a type. You can use **primitive type** names, such as int , float , or boolean . Or, you can use **reference types**, such as array, object, or interface names.

name - A member variable's unique name can be any legal identifier and, by convention, begins with a lowercase letter.

Member Method = *method declaration* and *body*

accessLevel – Control other classes' access to a method

static - declares method as a class not an instance method

abstract - method has no implementation and must be a member of an abstract class.

final - cannot be overridden by subclasses.

native – When we have external library of functions written in another language, such as C, you may use those functions from within Java

synchronize - Concurrently running threads often invoke methods that operate on the same data. Mark these methods with the synchronized keyword to ensure that the threads access information in a thread-safe manner.

Member Method = *method declaration* and *body*

returnType - Declare the data type of the value that it returns. If your method does not return a value, use the keyword void.

methodName - Method name can be any legal identifier.

(parameterList) - You pass information into a method through its arguments.

throws *exceptionList* - If your method throws any checked exceptions, your method declaration must indicate the type of those exceptions.

Naming your Methods

- Method names should be verbs and should be in mixed case.
  - **toString**
  - **compareTo**
  - **isDefined**
  - **setX**
  - **getX**
- A **method name should not be the same as the class name**, because constructors are named for the class. Typically, a method has a unique name within its class, but ...
- A method with the same signature and return type as a method in a superclass ***overrides*** or *hides* the superclass method.
- ***Name overloading*** for methods, which means that multiple methods in the same class can share the same name if they have different parameter lists.

Class Constructors

All classes have at least one constructor. A constructor is used to initialize a new object of that type and has the same name as the class.

```
public Stack()  {
        items =new Vector(10);
}
```

private protected public no specifier

A constructor has no return type. A constructor is called by the **new** operator, which automatically returns the newly created object.

```
public Stack(int initialSize)  {
        items =new Vector(initialSize);
}
```

Java **differentiates constructors** based on the number/type arg's.

Method/Constructor Arguments

- Arg's to any method or constructor is a comma-separated list of variable declarations, each variable is a pair of type/name.
- You can pass an argument of **any data type** into a method or a constructor –
  - **primitive data types** (doubles, floats, and int's etc.)
    - public void **getRGBColor**(int red,int green,int blue) {
      redValue = red; greenValue = green; blueValue = blue;
      }
  - **reference data types** (classes or arrays)
    - public static Polygon **polygonFrom**(Point [] **listOfPoints**)
      { // method accepts an array as an argument.
      // Java creates a new Polygon object and
      // initializes it from a **listOfPoints**
      }

The this keyword

- Within an instance method or a constructor, **this** is a reference to the *current object*—You can refer to any member of the current object from within an instance method or a constructor by using **this**.

```
public class HSBColor {
   private int  hue, saturation, brightness;
   public HSBColor (int hue, int saturation, int brightness)  {
      this.hue = hue;
      this.saturation = saturation;
      this.brightness = brightness;
}
```


Choosing the right access level

- If other programmers use your class, you want to ensure they do not misuse your members/class.
  - Start with the most restrictive access (private) level that makes sense for a particular member.
  - Avoid public member variables except for constants. Furthermore, if a member variable can be changed only by calling a method, you can notify other classes or objects of the change. Notification is impossible if you allow public access to a member variable. You might decide to grant public access if doing so gives you significant performance gains.
  - Limit the number of protected and package member variables.
  - If a member variable is a JavaBeans property, it must be private.

Instance vs. Class Members (revisited)

```
public class AClass {
   public int instanceInteger =0;
   public int instanceMethod(){return instanceInteger;}
   public static int classInteger =0;
   public static int classMethod(){ return classInteger; }

   public static void main (String [] args)) {
        AClass anInstance =  new AClass();
        AClass anotherInstance =  new AClass();
              //Refer to instance members through an instance.
        anInstance.instanceInteger =1;
        anotherInstance.instanceInteger =2;
```


Instance vs. Class Members (revisited)

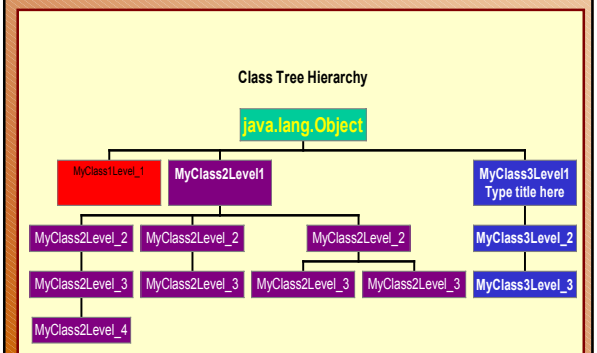
```
   System.out.println(anInstance.instanceMethod());
   System.out.println(anotherInstance.instanceMethod());
        //  Illegal to refer directly to instance members from a class method
        //        System.out.println(instanceMethod()); // illegal
        //        System.out.println(instanceInteger);   // illegal
        //        Refer to class members through the class...
   AClass.classInteger =7;
   System.out.println(classMethod());
        // ...    or refer to class members through an instance.
   System.out.println(anInstance.classMethod());
   //Instances share class variables
   anInstance.classInteger =9;
   System.out.println(anInstance.classMethod());
   System.out.println(anotherInstance.classMethod());
} }
```

Out
1
2
7
7
9
9

Instance vs. Class Members (revisited)

- In general, member declared within a class is an **instance member**, and you can access an instance member and call an instance method only through a reference to the instance [anInstance.instanceMethod()].
- For a **class variable**, which is declared by using the **static** modifier, the runtime system allocates a class variable once per class, regardless of the number of instances created of that class.
- All instances of a class share the same copy of the class's class variables. You can access class variables either through an instance or through the class itself. Similarly, class methods can be invoked on the class or through an instance reference. Note that when the program changes the value of classVariable , its value changes for all instances.

Inheritance



Inheritance

- The **Object** class, defined in the java.lang package, is the most general class & defines and implements behavior that every class needs. All other classes derive from Object , many classes derive from those classes, and so on, forming a hierarchy of classes.
- Classes at the bottom of the hierarchy are more specialized. A ***subclass*** derives from its ***superclass*** (direct ancestor – descendant organization).
- Every class has one and only one immediate superclass. A subclass inherits all the member variables and methods from its superclass. But, the subclass has no access to private inherited members.
- Constructors are not members and so are not inherited by subclasses.

Overriding methods

- An instance method in a **subclass** with the same signature and return type as an instance method in the **superclass** *overrides* the superclass's method. This allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed.
- Ex: **java.lang.Object** contains **toString().** Every class inherits this method. The implementation in Object is not very useful for all subclasses;

```
public class MyClass {   //Overrides toString in Object
        private int anInt =4;
        public String toString(){
          return "Instance of MyClass.anInt ="+anInt;
} }
```

Using super

- If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of **super** .

```
public class Superclass {
   public boolean aVariable;
   public void aMethod(){  aVariable =true;  }
}
public class Subclass extends Superclass {
   public boolean aVariable;
   public void aMethod() { //overrides aMethod
        aVariable =false;   super.aMethod();
        System.out.println(aVariable);
        System.out.println(super.aVariable );
}}
```

Out
false
true

Using super

- You can also use super within a constructor to invoke a superclass's constructor.

```
class AnimationThread extends Thread {
     int framesPerSecond;
     int numImages;
     Image [] images;
     public AnimationThread(int fps,int num) {
          super("AnimationThread");
          this.framesPerSecond =fps;
          this.numImages =num;
          this.images =new Image [numImages ];
          …
  }  }
```


Subclasses of java.lang.Object

- Every class is a descendant, direct or indirect, of the Object class. This class defines the basic state and behavior that all objects must have, such as the ability to compare oneself to another object, to convert to a string, to wait on a condition variable, to notify other objects that a condition variable has changed, and to return the class of the object.
  - **clone**
  - **equals and hashCode**
  - **finalize**
  - **toString**
  - **getClass**
  - **notify**
  - **notifyAll , and**
  - **wait**

Final Classes and Methods

- A final class cannot be subclassed. Two reasons:
  - ***Security***: to increase system security by preventing system subversion. To subvert systems hackers often create a subclass of a class and then substitute the subclass for the original. The subclass looks and feels like the original class but does vastly different things, possibly causing damage or getting due to possible overriding.
  - ***Design***: for reasons of good object-oriented design. If your class is *perfect* or that, conceptually, your class should have no subclasses.

```
final class ChessAlgorithm {  ...  }
```


Abstract Classes and Methods

- If a class represents an *abstract concept* it should not be instantiated. Take, for example, food in the real world. Have you ever seen an instance of food? No. What you see instead are instances of carrot, apple, and chocolate chip cookies.
- It makes sense to create an abstract **Number** object. A class such as **Number**, which represents an abstract concept and should not be instantiated, is called an ***abstract class***. An abstract class can only be subclassed; it cannot be instantiated.

```
abstract class Number {
...
}
```


Abstract Classes and Methods

- An abstract class can contain ***abstract methods***—methods with no implementation. In practice, abstract classes provide a complete or partial implementation of at least one method. If an abstract class contains only abstract method declarations, it should be implemented as an interface instead.
- Ex: In an object-oriented drawing application, you can draw circles, rectangles, lines, Bézier curves, and so on. These graphic objects all have certain states (position, bounding box) and behaviors (move, resize, draw) in common. You can take advantage of these similarities and declare them all to inherit from the same parent object—for example, GraphicObject.

Abstract Classes and Methods

```
abstract class GraphicObject {
        int x, y;
        ...
        void moveTo(int newX, int newY) { … }
        abstract void draw();
}
class Circle extends GraphicObject {
        void draw(){ ... }
}
class Rectangle extends GraphicObject {
        void draw(){ ... }
}
```

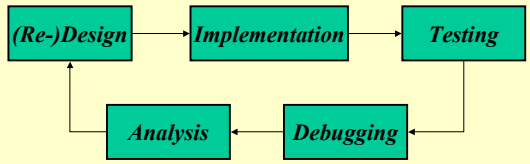

Nested Classes

- A nested class is a member of another class.

```
class EnclosingClass {
        ...
        class ANestedClass {
                ...
        }
}
```

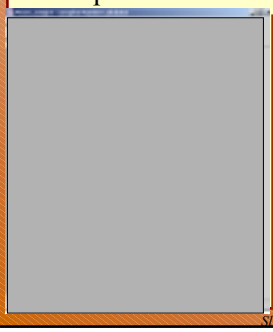
- You use **nested classes to reflect and to enforce the relationship between two classes**. You should define a class within another class when the nested class makes sense only in the context of its enclosing class or when it relies on the enclosing class for its function. Ex, a text cursor might make sense only in the context of a text component.

Project Organization



Project Design

- Complex Number Calculator Example:



- ***Runnable as an Applet and an Application***
- ***Identify Complex-Number Operations***
- **State of the application:**
  - *What widgets (components) do we need to get user input?*
  - *How to present the result to the user as output?*
  - *How to arrange all components in the frame?*
- **Behavior of the application**
  - *Clean/resize/change appearance*

Project Design

- Complex Number Calculator Example:

- ***Runnable as an Applet and an Application***
- ***Identify Complex-Number Operations***
- **State of the application:**
  - *What widgets (components) do we need to get user input?*
  - *How to present the result to the user as output?*
  - *How to arrange all components in the frame?*
- **Behavior of the application**
  - *Clean/resize/change appearance*

Project Design

- Complex Number Calculator Example:

- ***Applet vs. Application – write an applet, then insert main()***
- ***Operations {-, +. *, /, ||, 1/z, Clear}***
- **State:**
  - *JTextField, Jlist, Jpanel, JScrollPane, JLabel*
  - *ComplexDrawPanel() – private class*
  - *BorderLayout*
- **Behavior:**
  - *Action triggered by List element selection (arithmetic operation)*
  - *Read in JTextField strings*
  - *Compute result*
  - *Draw all Complex numbers*
  - *Scale DrawingPanel based on |z|*

Project Design

- Complex Number Calculator Example:

- **Identify specific members:**
  - Constructors
  - Methods
  - Variables
  - Private classes
- Write empty bodies for most methods you need
- Try to compile, fix little bugs.
- Look and feel of the applet?!?
- Begin putting in details about the state and behavior of the application
- Compile, test, debug, redesign, re-implement, recompile etc.
- Final tests, validation, documentation and project submission