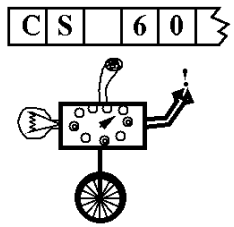


More on Classes

October 12, 2001

Review: Aggregation (has-a)

```
class Point {
   private int x, y;
   Point(int x, int y)     { this.x = x; this.y = y }
   void move(int x, int y) { this.x = x; this.y = y }
   int getx() { return x; }
   int gety() { return y; }
}

class Shape {
   Point center; // every Shape references a Point
   Shape(int x, int y) { center = new Point(x,y); }
   void move(int x, int y) { center.move(x,y); }
   void draw() {};
   void moveAndDraw(int x, int y)
       {this.move(x,y); this.draw(); }
}
```

Review: Inheritance (is-a)

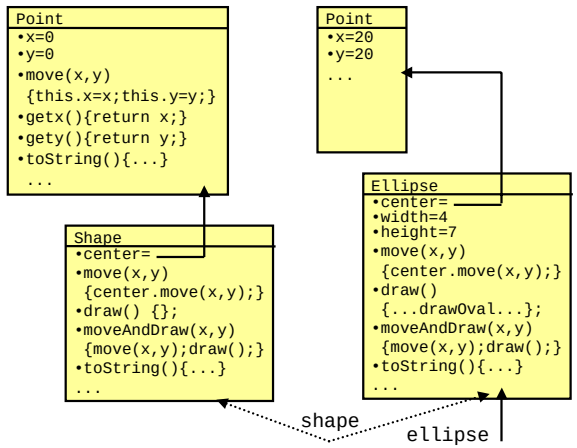
```
class Square extends Shape {
   int size;
   Square(int x, int y, int size)
        { super(x,y); this.size = size }
   void draw() { ...drawRect... }
}

class Ellipse extends Shape {
   int width, height;
   Ellipse(int x, int y, int w, int h) {
        super(x,y); width = w; height = h;
   }
   void draw() { ...drawOval... }
}
```

What Code Runs At Each Call?

```
Shape   shape = new Shape(0,0);
Square  square = new Square(10,10,5);
Ellipse ellipse = new Ellipse(20,20,4,7);

ellipse.move(14,12);
ellipse.moveAndDraw(12,14);

shape = ellipse;
shape.draw();
shape.moveAndDraw(2,2);

shape = square;
shape.moveAndDraw(42,42);
```

Dynamic dispatch strikes again!

Pictorially…



(from Joshua Bloch, *Effective Java)*

Can You Figure Out The Answer?

```
class InstrumentedHashSet extends HashSet {
   private int addCount = 0;  // count of elements added
   public InstrumentedHashSet() {}
   public InstrumentedHashSet(Collection c) { super(c); }
   public boolean add(Object o) {
        addcount++;
        return super.add(o);
   }
   public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
   }
   public int getAddCount() { return addCount; }
}
...
InstrumentedHashSet ihs = new InstrumentedHashSet();
ihs.addAll(Arrays.asList(new String[]{"A","B","C"}))
```

Pitfalls of Overriding

- Overriding breaks encapsulation
  - To do overriding right, need to know details about the *implementation* of the superclass(es!)
    - Not just its interface!
    - Full knowledge of which methods call which other methods, and when.
    - What if the implementation of the superclass changes?
  - Some people suggest using only extension unless absolutely necessary.
- Still, there are a few cases where it's safe and useful.
  - Where superclass is designed for overriding

Methods of `java.util.Stack`

- Methods of Stack proper:
  - boolean empty()
  - Object peek()
  - Object pop()
  - Object push(Object item)
  - int search(Object o)
- Methods of Vector:
  - add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll, copyInto, elementAt, elements, ensureCapacity, equals, firstElement, get, hashCode, indexOf, insertElementAt, isEmpty, lastElement, lastIndexOf, remove, removeAll, removeAllElements, removeElement, removeElementAt, removeRange, retainAll, set, setElementAt, setSize, size, subList, toArray, toString, trimToSize
- Methods of AbstractList:
  - iterator, listIterator
- Methods of Object (not otherwise over-ridden)
  - finalize, getClass, notify, notifyAll, wait

To understand all methods for class
`java.util.Stack`
you need to understand:
`java.util.Vector`
`java.util.AbstractList`
`java.util.AbstractCollection`
`java.lang.Object`

(See example in text, section 7.12)

Safe Overriding

- The Applet class includes the following methods, among others:

```
public void    init()
public void    start()
public boolean mouseDown(Event e, int x, int y)
public boolean mouseDrag(Event e, int x, int y)
public boolean mouseUp  (Event e, int x, int y)
public boolean mouseMove(Event e, int x, int y)
public void    update(Graphics g)
public void    paint(Graphics g)
```

- The Applet design
  - Promises exactly when these methods will be called
  - Explains the default implementation, so you know whether you want to override it.
  - Applets that want to know about mouse clicks can override `mouseDown`; applets that just draw trees can leave the default implementation unchanged (do nothing)

Example: EZdraw.java (extends Applet)

```
/* remember the current mouse coordinates */
  private int last_x;
  private int last_y;
  private void remember(int x, int y)
    {
    last_x = x; last_y = y;
    }

  /** mouseDown is called when user presses the mouse to start a drawing */

  public boolean mouseDown(Event e, int x, int y)
    {
    remember(x, y);
    return true;
    }

  /** mouseDrag is called when user drags the mouse */

  public boolean mouseDrag(Event e, int x, int y)
    {
    Graphics g = getGraphics();
    g.drawLine(last_x, last_y, x, y);
    remember(x, y);
    return true;
    }
```

These **must** be declared with same **type** and **visibility** as in base class!

Abstract Classes

- An *abstract* class is a class in which certain methods are left unimplemented.
  - Cannot create objects of this class; they'd be incomplete.
    - Can still have variables of this class, though!
  - To be useful, need to create subclasses that implement the missing methods.

```
abstract class Shape {
   Point center; // every Shape references a Point
   Shape(int x, int y) { center = new Point(x,y); }
   void move(int x, int y) { center.move(x,y); }
   abstract void draw();
   void moveAndDraw(int x, int y) {move(x,y); draw(); }
}
```

Abstract Classes vs. Interfaces

- Somewhat similar concepts
  - C++ has only the former (which get used as interfaces).
  - Both specify a collection of methods
- Differences?

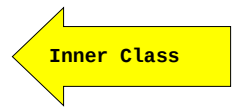
Inner Classes

- In Java (and C++), classes can be nested within one another.
- Objects in the inner class has available instance variables and methods of the outer class.

Ways to Construct ClosedList

```
class Cell {…}
class ClosedList {…}
```

```
class ClosedList
{
  class Cell {…}
  ...
}
```



Usage

- Normally one or more objects of the inner class are created for a given object of the outer class.
- Objects of the inner class only make sense in the context of a supporting object of the outer class.

Example: `Iterator`

- We want to define an `Iterator` for a `ClosedList`.
  - For read-only iteration, the `Iterator` class can be defined outside the `ClosedList` class.
  - For modification, such as `remove()`, it is sometimes necessary to *change* part of the corresponding ClosedList object.
- By making the `Iterator` an inner class, we can:
  - Use data elements defined in the `ClosedList`.
  - Avoid exposing those data elements to the world at large.
  - Use iterators outside `ClosedList`

ClosedList.Iterator

```
class ClosedList
{
private Cell head;
private Cell tail;
…
public Iterator getIterator()
  {
  return new Iterator(head);
  }

public class Iterator implements java.util.Iterator
  {
  private Cell current;  // cell with value to be
                         //    returned next
  private Cell previous;

  public Iterator(Cell head)
    {
    current = head;
    previous = null;
    }
    ...
```

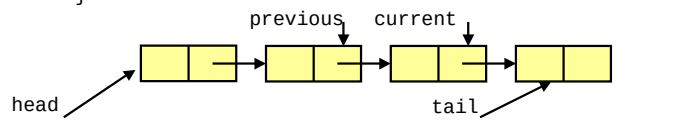
Hidden from the outside!

ClosedList.Iterator: remove()

Defined to remove the value just produced by next().

```
public void remove()
   {
   if( previous == null )
     {
     head = head.getNext();
     }
   else
     {
     previous.setData(current.getData());  // reuse
     previous.setNext(current.getNext());  // previous
     current = previous;                   // lose current
     }
   }
```

head is defined in outer class!



Aside: Converting a list to a String (e.g. for printing entire list at once)

```
public String toString()
  {
  StringBuffer buffer = new StringBuffer();
  Iterator it = new Iterator(head);
  if( it.hasNext() )
    {
    buffer.append(it.next()); // first element
    }

  while( it.hasNext() )
    {
    buffer.append(" " + it.next()); // leave space
    }
  return buffer.toString();
  }
```

Handling by Inner Classes

- Applet defines inner classes for various types of handling.
- Applet adds handlers at initialization.

Example: SoSodraw.java (extends Applet)

```
import java.awt.event.*
public void init()
    {
    ...
    addMouseListener(new SoSoMouseListener());
    addMouseMotionListener(new SoSoMouseMotionListener());
    }

class SoSoMouseListener implements MouseListener
  {
  /* mousePressed is called when user presses the mouse */

  public void mousePressed(MouseEvent e)
    {
    int x = e.getX();
    int y = e.getY();
    remember(x, y);
    }
```

Must also define, even if you don't use them:

```
public void mouseClicked(MouseEvent e);
public void mouseEntered(MouseEvent e)
public void mouseExited(MouseEvent e)
public void mouseReleased(MouseEvent e)
```

because the **interface** MouseListener does.

Example: SoSodraw.java (2)

```
class SoSoMouseMotionListener implements MouseMotionListener
  {
  /** mouseDrag is called when user drags the mouse */

    public void mouseDragged(MouseEvent e)
      {
      int x = e.getX();
      int y = e.getY();
      Graphics g = getGraphics();
      g.drawLine(last_x, last_y, x, y);
      remember(x, y);
      }

  public void mouseMoved(MouseEvent e)
    {
    }
  }
```

Simplification: Adapters

```
import java.awt.event.*
public void init() { ...
    addMouseListener(new SoSoMouseListener());
    addMouseMotionListener(new SoSoMouseMotionListener());
    }

class SoSoMouseListener  extends MouseAdapter {
  public void mousePressed(MouseEvent e) {
    int x = e.getX(); int y = e.getY();
    remember(x, y);
    }
}
class SoSoMouseMotionListener extends MouseMotionAdapter {
    public void mouseDragged(MouseEvent e) {
      int x = e.getX(); int y = e.getY();
      Graphics g = getGraphics();
      g.drawLine(last_x, last_y, x, y);
      remember(x, y);
      }
}
```

Override only what you care about

Override only what you care about