CS1007: Object Oriented Design and Programming in Java

Lecture #8
Oct 4

Shlomo Hershkop
shlomo@cs.columbia.edu

Outline

- Unit Testing
- Sorting Algorithms
- Polymorphism
- Interfaces
- Basic graphics
- Layout managers
- Anonymous Classes

Feedback

- From lab experience
  - Please see us if you need more help with anything in Java
  - Any comments?

HW2

- Help you work with UML and design specs
- More practical java work

- Out Today
- Programming Due: Oct 17 11:59pm.
- Written Part Due: Class Oct 18

- Note: For those who missed handing in written part 1, see TA

3 proposals in Testing

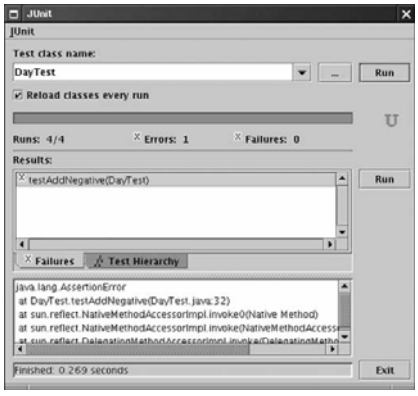
- Don't:
  - Hope to be bought out before anyone realizes
    - Requirements: ticket to get out of town once they realize
- Assemble everything and then test
  - See above
- Test individual components before integrating
  - Ability to only test each piece separately, but allows you to work out many bugs early on.

Unit Testing

- Unit test = test of a single class
- Design test cases during implementation
- Run tests after every implementation change
- When you find a bug, add a test case that catches it

JUnit

- http://www.junit.org/
- Framework for running tests on your code
- Need to plan out tests not random
  - Code a special class to run tests on your other classes
  - Will explore this in next lab
- Need to realize advantages and disadvantages of this framework
  - It is only a TOOL!



JUnit

- Test class name = tested class name + Test
- Test methods start with test

```
import junit.framework.*;
public class DayTest extends TestCase
{
   public void testAdd() { ... }
   public void testDaysBetween() { ... }
   . . .
}
```

JUnit

- Each test case ends with assertion
- Test framework catches assertion failures

```
public void testAdd()
{
   Day d1 = new Day(1970, 1, 1);
   int n = 1000;
   Day d2 = d1.addDays(n);
   assertTrue(d2.daysFrom(d1) == n);
}
```

Shift gears

- Sorting integers

Sort

- A sorting algorithm takes an unordered array of objects and returns an array of objects in ascending or descending order.
- Ascending is defined by the programmer or object type.
- We can use integers as an example
  - Number line defines order

Bubble Sort

- The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required.
- The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order).
- This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list.

Algorithm

- Input: List of integers $X_0, X_1 \ldots X_n$
- Output: ascending order

1. i = 0, swap =0
2. If $X_i > X_{i+1}$ swap X_i and X_{i+1} swap =1
3. i = i + 1
4. if i < n goto step 2
5. If swap > 0 goto step 1
6. Return sorted list

Pseudo code

```
void bubbleSort(int numbers[], int array_size)
{
  int i, j, temp;

  for (i = (array_size - 1); i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      if (numbers[j-1] > numbers[j])
      {
        temp = numbers[j-1];
        numbers[j-1] = numbers[j];
        numbers[j] = temp;
      }
    }
  }
}
```

Example

- 6 9 3 1 8

How to analyze this algorithm

- Will be taught in data structures
- Enough to know:
  - Slowest sort in general
  - Run time:
    - Anyone know how many comparisons required?
  - Advantages:
    - For small number of items, ok to use
    - Simple

Polymorphism

- Definition:
  - Programming language's ability to process objects independent of their data type or class with the same set of code
  - i.e. example draw a shape on the screen
    - Triangle
    - Square
    - Circle
  - Perfect for Object Oriented design

Interface

- Java interface defines methods which need to be implemented by anyone implementing the interface
- No implementation
- Implementing class must supply implementation of all methods

The Icon Interface Type

```
public interface Icon
{
 int getIconWidth();
 int getIconHeight();
 void paintIcon(Component c,
  Graphics g, int x, int y)
}
```

Example

- Use JOptionPane to display message:

```
JOptionPane.showMessageDialog(null, "Hello,
   World!");
```

- Note icon to the left

images

- Can specify arbitrary image file

```
JOptionPane.showMessageDialog(
 null,
 "Hello, World!",
 "Message",
 JOptionPane.INFORMATION_MESSAGE,
 new ImageIcon("globe.gif"));
```

Displaying an Image

- What if we don't want to generate an image file?
- Fortunately, can use any class that implements Icon interface type
- ImageIcon is one such class
- Easy to supply your own class

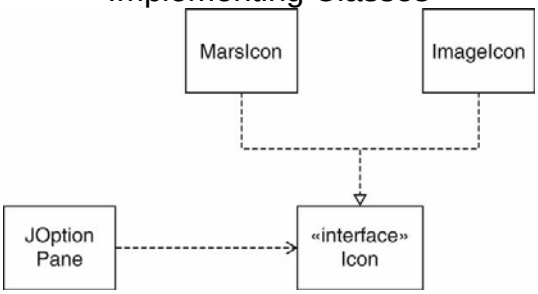
Marsicon.java

```
import java.awt.*;
02: import java.awt.geom.*;
03: import javax.swing.*;
04:
05: /**
06:    An icon that has the shape of the
   planet Mars.
07: */
08: public class MarsIcon implements Icon
09: {
10:    /**
11:    Constructs a Mars icon of a given
   size.
12:    @param aSize the size of the icon
13:    */
14:    public MarsIcon(int aSize)
15:    {
16:       size = aSize;
17:    }
18:
19:    public int getIconWidth()
20:    {
21:       return size;
22:    }
23:
24:    public int getIconHeight()
25:    {
26:       return size;
27:    }
28:
29:    public void paintIcon(Component c,
   Graphics g, int x, int y)
30:    {
31:       Graphics2D g2 = (Graphics2D) g;
32:       Ellipse2D.Double planet = new
   Ellipse2D.Double(x, y,
33:             size, size);
34:       g2.setColor(Color.RED);
35:       g2.fill(planet);
36:    }
37:
38:    private int size;
39: }
```

Using it

```
01: import javax.swing.*;
02:
03: public class IconTester
04: {
05:    public static void main(String[] args)
06:    {
07:       JOptionPane.showMessageDialog(
08:             null,
09:             "Hello, Mars!",
10:             "Message",
11:             JOptionPane.INFORMATION_MESSAGE,
12:             new MarsIcon(50));
13:       System.exit(0);
14:    }
15: }
16:
```

The Icon Interface Type and Implementing Classes



Polymorphism

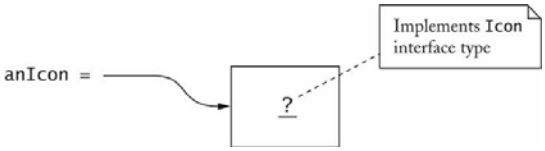
- `public static void showMessageDialog(...Icon anIcon)`
- `showMessageDialog` shows
  - icon
  - message
  - OK button
- `showMessageDialog` must compute size of dialog
- width = icon width + message size + blank size
- How do we know the icon width?

```
int width = anIcon.getIconWidth();
```

PolyMorphism

- showMessageDialog doesn't know which icon is passed
  - ImageIcon?
  - MarsIcon?
  - . . .?
- The actual type of anIcon is not `Icon`
- There are no objects of type `Icon`
- `anIcon` belongs to a class that implements `Icon`
- That class defines a `getIconWidth` method

A Variable of Interface Type



So…

- Which getIconWidth method is called?
- Could be
  - MarsIcon.getIconWidth
  - ImageIcon.getIconWidth
  - . . .
- Depends on object to which anIcon reference points, e.g.

```
showMessageDialog(..., new MarsIcon(50))
```

- Polymorphism: Select different methods according to actual object type

Benefits

- Stronger OO Design
- Loose coupling
  - showMessageDialog decoupled from ImageIcon
  - Doesn't need to know about image processing
- Extensibility
  - Client can supply new icon types

The `Comparable` Interface Type

- Collections has static sort method:

```
ArrayList<E> a = . . .
Collections.sort(a);
```

- Objects in list must implement the `Comparable` interface type

```
public interface Comparable<T>
{
 int compareTo(T other);
}
```

- Interface is parameterized (like `ArrayList`)
- Type parameter is type of `other`

- `object1.compareTo(object2)` returns
  - Negative number if `object1` less than `object2`
  - 0 if objects identical
  - Positive number if `object1` greater than `object2`

- sort method compares and rearranges elements

```
if (object1.compareTo(object2) > 0) . . .
```

- `String` class implements `Comparable<String>` interface type: lexicographic (dictionary) order
- `Country` class: compare countries by area

```
01: /**
02:    A country with a name and area.
03: */
04: public class Country implements Comparable<Country>
05: {
06:    /**
07:       Constructs a country.
08:       @param aName the name of the country
09:       @param anArea the area of the country
10:    */
11:    public Country(String aName, double anArea)
12:    {
13:       name = aName;
14:       area = anArea;
15:    }
16:
```

testing

```
01: import java.util.*;
02:
03: public class CountrySortTester
04: {
05:    public static void main(String[] args)
06:    {
07:       ArrayList<Country> countries = new
   ArrayList<Country>();
08:       countries.add(new Country("Uruguay", 176220));
09:       countries.add(new Country("Thailand", 514000));
10:       countries.add(new Country("Belgium", 30510));
11:
12:       Collections.sort(countries);
13:       // Now the array list is sorted by area
14:       for (Country c : countries)
15:          System.out.println(c.getName() + " " + c.getArea());
16:    }
17: }
```

The Comparator interface type

- How can we sort countries by name?
- Can't implement Comparable twice!
- Comparator interface type gives added flexibility

```
public interface Comparator<T>
{
   int compare(T obj1, T obj2);
}
```

- Pass comparator object to sort:

```
Collections.sort(list, comp);
```

- Comparator object is a function object
- This particular comparator object has no state
- State can be useful, e.g. flag to sort in ascending or descending order

The Comparator interface type

```
01: import java.util.*;
02:
03: public class CountryComparatorByName implements
  Comparator<Country>
04: {
05:    public int compare(Country country1, Country
  country2)
06:    {
07:       return
  country1.getName().compareTo(country2.getName());
08:    }
09:
10: }
```

```
01: import java.util.*;
02:
03: public class ComparatorTester
04: {
05:    public static void main(String[] args)
06:    {
07:       ArrayList<Country> countries = new ArrayList<Country>();
08:       countries.add(new Country("Uruguay", 176220));
09:       countries.add(new Country("Thailand", 514000));
10:       countries.add(new Country("Belgium", 30510));
11:       Comparator<Country>  comp = new CountryComparatorByName();
12:       Collections.sort(countries, comp);
13:          // Now the array list is sorted by area
14:       for (Country c : countries)
15:         System.out.println(c.getName() + " " + c.getArea());
16:    }
17: }
18:
```

Anonymous Classes

- No need to name objects that are used only once

```
Collections.sort(countries,
   new CountryComparatorByName());
```

- No need to name classes that are used only once

```
Comparator<Country> comp = new
   Comparator<Country>()
   {
      public int compare(Country country1, Country country2)
      {
         return country1.getName().compareTo(country2.getName());
      }
   };
```

- anonymous new expression:
  - defines anonymous class that implements Comparator
  - defines compare method of that class
  - constructs one object of that class
- Cryptic syntax for very useful feature

Anonymous Classes

- Commonly used in factory methods:

```
public static Comparator<Country> comparatorByName()
{
   return new Comparator<Country>()
   {
      public int compare(Country country1, Country
  country2)

      { . . . }
   };
}
```

- Collections.sort(a, Country.comparatorByName());
- Neat arrangement if multiple comparators make sense (by name, by area, ...)

Next Time

- Continue reading
- Meet Thursday in **Class**
- Will continue with more analysis of the graphics components of awt.

- Continue reading book