Objectives

- Understand Applet model
- Understand Exceptions
  - throw-catch block
  - rethrowing

Applets

- `import java.awt.*;`
- `import java.applet.*;`
- Superclass
  - No need to implement main()
  - public class Minimal extends Applet {
  

Model in mind

- **Controller**
  - keeps track of your applet and its graphics
  - monitoring events, like mouse clicks, window motion, etc.
  - if there is a need to repaint your screen, it calls the paint method on your applet object
  - if there is a mouse motion (mouse click, mouse in, mouse out) calls the appropriate methods on your applet object

Model (Good enough)

```java
method controller() {
    registers applet and its graphics objects
    loop {
        if need to repaint
            for all registered objects
                object.paint(g);
        if mouse Down
            for all registered objects
                object.mouseDown(x, y);
        if mouseUp
            etc.
    } // forever or until stop
```

Applet responsibilities

- `init()` – read external images, set up the graphics, buttons, text, etc.
- `paint(Graphic g)` – modify the Graphic object
- `mouseClick(Event e, int x, int y)` – you can ignore the Event object if you want, x and y are obvious

Problem

- load an image into an applet
- mouse over the image
- determine the color of the pixel under the mouse?
public void init()

public class test extends Applet {
    Image b;
    public void init()
    {
        b = getImage(getDocumentBase(), "rocketship.gif");
    }
}

public void repaint(Graphics g)
{
    g.drawImage(b,0,0,this);
}

mouse events?

public boolean mouseDown(Event f,int x,int y)
{
    System.out.println("pixel x = " + x + "pixel y = " + y);
    repaint();
}
Exceptions – why?

- Logic errors
  - array bounds problems
- Users
  - don’t know correct input
- Devices
  - are not always predictable
  - files, e.g., move

Without Exceptions

- need to set globals
- pass extra variables
- hard to understand
- logic for handling exceptions mixed into problem solving code

Exception objects to the rescue!

- Exception object describes problem
- Exception handler invoked to analyze exception object and take action

3 things to understand

- representation
- definition of the handler
- control flow
Exception representation

- Derives from Throwable objects
  - Throwable()
  - Throwable (String message)
  - String getMessage()
  - void printStackTrace()

Exception representation

- Unchecked exceptions – system errors or runtime errors – low memory - you won’t handle these
- Checked exceptions – things you or some other class throw
- Catch or Throw policy
  - networking
  - Streams, etc. - pain

Creating an exception

- new myException(" oops")

throwing and catching

if a problem
throw new MyException("oops");
throwing and catching

```java
public class myClass {
    ...
    public int myMethod() throws MyException {
        if a problem
            throw new MyException("oops");
        else
            return 1;
    }
}
```

catching

```java
int b;
MyClass mc = new MyClass();
b = mc.myMethod();
```

catching

```java
int b;
MyClass mc = new MyClass();
try {
    b = mc.myMethod();
} catch (Exception e) {
    throw e;
}
```
return to applet

Convert the image to a pixel map in init()
...
int a[] = new int[myHeight*myLength];
r = new PixelGrabber(b,0,0,myLength,myHeight,a,0,myLength);
try {
    r.grabPixels();
} catch (InterruptedException e) {
    System.out.println("problem in grabPixels");
}
...

get pixel in mouse event

// get the pixel where the mouse is
int pixelcolor = a[y*myLength + x];
System.out.println("pixelcolor = " +
pixelcolor);
For week after break

- Read Chapter 4
- Lab on Monday of return uses this code

Objectives

- Enhance your understanding of program design methodology
- Give insights into following stages:
  - problem requirements
  - problem specification
  - program design
  - program implementation and testing

Objectives

- Enhance understanding of specialization inheritance
- Enhance understanding of specification inheritance
- Enhance understanding of role specification inheritance plays in event-based programming
Stages

- Problem requirements
- problem specifications
- software design
- identifying classes
- implementation
- testing

Requirements

- Heating simulation
  - thermostat
  - furnace
  - living area
  - environment (outside the room)

Problem requirements

- Room
  - initial temp, time T=0
  - Determine room temp at T+Δ
    - heat from furnace and loss from environment
  - Operations
    - turn on if temp too low and furnace off
    - turn off if temp too high and furnace on
    - do nothing
- Rinse and repeat

Problem Specifications

- Input/Output document
  - exactly what output produced for EVERY possible input
**Inputs**

- Started from the command line
  - `java HeatingSimulation parameters...`
- `in` = initial temperature (72.0)
- `out` = outside environment (50)
- `set` = thermo setting (72)
- `cap` = capacity (8500 BTU/hour)
- `eff` = efficiency (.95)
- `size` = size in sq. ft. (250)
- `freq` = number of ticks between output (5)
- `length` = seconds to run (7200)

**Execution**

- `java HeatingSimulation size=1000 in=65`

**Problem spec on p. 156**

- model 4 components
  - room of size x, init temp
  - thermo – setting
  - furnace – cap and eff
  - environment
  - clock
  - Δ = how far to advance clock
  - determine new temp using formulas
  - check thermostat
  - adjust furnace accordingly
  - produce a line of output

**Software Design**

- id the classes
  - state
  - behavior
  - diagram
Problem spec on p. 156

- model 4 components
  - room of size x, init temp
  - thermo - setting
  - furnace - cap and eff
  - environment
- clock
  - Δt - how far to advance clock
  - determine new temp using formulas
  - check thermostat
  - produce a line of output

See book for classes and behaviors
Use of inheritance

- Specialization
  - lots of furnaces
  - gas, electric, etc.
  - could just include lots of special case code for furnace – pilot light on, etc.
- let’s abstract and localize
- superclass Furnace
  - shared state and common behavior
  - add new behavior
  - override existing ones

Use of interfaces

- How do we know when to update?
- One scenario
  - Room checks clock
  - Thermostat checks clock
  
  Loop {
  clock.update();
  room.update();
  thermostat.update();
  } until time up

Use of interfaces

- Better way
  - clock sends messages
  - all we have to do is make sure interested parties have a listener method
- Use Interfaces – ClockListener
  - public interface ClockListener {
    void preEvent(double timeInterval);
    void event();
  }

PreEvent() and event()

- Commonly used technique
  - preEvent – does a calculation based on state of simulation
    - room – check on/off of furnace, check outdoor temp, solve heat loss and gain formulas
    - Don’t update state
  - event() – update state
Example – Scissors, paper, rock

// method to check
1. check my status (scissor, paper or rock)
2. check other status (scissor, paper or rock)
3. calculate who wins
4. print out winner
5. change to new status

Ordering problem
Rock guy goes first

Ordering problem
Paper gal goes

PreEvent
I Win
Next I'll be paper

I Lose
Next I'll be scissors
Interfaces

- Makes it easy to add new class
  - they just need to implement ClockListener interface
- To set up the simulation
  - create a clock
  - create room, furnace, etc.
  - clock.addListener(room)
  - clock.addListener(furnace)
Next time

- Read 4.5, 4.6, 4.7
- In particular, while reading 4.5, think about unit tests you might write