(*Lec 02*) Programming Aside: Java[tm]

- What you know
  - C/C++ programming
  - Probably some object-oriented design issues
  - Maybe already some Java (if so, this is review…)
- What you don't know
  - Java
    - Latest, greatest entrant in the language-wars
    - Subject of significant interest, investment, and hype
  - What we want to do here
    - Talk about the features in the language
    - Get you some basic familiarity
    - Show some examples
    - Do 760 Project 1 in Java

Copyright Notice

© Rob A. Rutenbar, 2001
All rights reserved.

You may not make copies of this material in any form without my express permission.

Handouts

- Physical
  - Lecture 02 -- Java review
- Electronic
  - Nothing today

Where Are We?

- Doing some JAVA background you need for Project 1...

	M	T	W	Th	F	
Aug	27	28	29	30	31	1
Sep	3	4	5	6	7	2
	10	11	12	13	14	3
	17	18	19	20	21	4
	24	25	26	27	28	5
Oct	1	2	3	4	5	6
	8	9	10	11	12	7
	15	16	17	18	19	8
	22	23	24	25	26	9
	29	30	31	1	2	10
Nov	5	6	7	8	9	11
	12	13	14	15	16	12
Thnxgive	19	20	21	22	23	13
	26	27	28	29	30	14
Dec	3	4	5	6	7	15
	10	11	12	13	14	16

Introduction
Advanced Boolean algebra
JAVA Review
Formal verification
2-Level logic synthesis
Multi-level logic synthesis
Technology mapping
Placement
Routing
Static timing analysis
Electrical timing analysis
Geometric data structs & apps

Java -- Good References

- Two book suggestions
  - David Flanagan, *JAVA in a Nutshell: A Desktop Reference Guide*, O'Reilly, 2nd Edition, May 1997.

    A good nuts and bolts reference with a lot of emphasis on how Java differs from C and from C++.

  - Mary Campione and Kathy Walrath, *The Java Tutorial: Object-Oriented Programming for the Internet*, (The JAVA Series), Addison Wesley, 1996.

    Another good treatment from some Java folks at SUN, with good intro stuff and lots of focus on network and internet-centric stuff.

Java -- Good References

- Web references on line
  - http://www.javasoft.com
    The SUN main site for Java. You can see product info, download free Java code, etc.
  - http://java.sun.com/docs/books/tutorial/
    The Campione & Walrath book, essentially all the tutorials ON LINE, with examples. Called "The JAVA^{tm} Tutorial" page…
  - http://java.sun.com/j2se/
    Where to look for a browser for all the Java classes, objects, methods, etc., that you use to bolt components together to make programs; this is the recent version(s) of "the JAVA 2 Platform"
  - http://www.gamelan.com
    A useful directory for Java code examples, a good place to snoop for Java code you can borrow/use.

JAVA -- What Is It?

- A programming language developed by SUN
  - Originally developed as a language for "set top boxes" ie, for boxes that let TVs behave like computers.
  - Redirected to be "an internet language" when this didn't pan out
  - Released in 1995
  - Development led by James Gosling -- the CMU CS alum who wrote the original "emacs" editor as a PhD student here
- Why does anybody care…?
  - It's a very pretty, elegant language
  - It specifically targets the "internetworked" world
  - It's being marketed very aggressively
  - It was regarded as a challenge to the "market domination" of the Microsoft/Intel duopoly, when it first appeared.

Java -- Big Picture

- SUN says this

  [Java is a] simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language
- Details
  - *Simple*: less syntax than C++ or even C; very clean.
  - *Object-oriented*: from the ground up, unlike C++ where it's add-on.
  - *Distributed*: lots of direct support for networked environment.
  - *Interpreted*: you get a Java Virtual Machine to run your code.
  - *Robust*: since it's simple & interpreted, some errors you can't make.
  - *Secure*: again, since it's interpreted. Also some design features.
  - *Architecture neutral*: since it's interpreted, it runs just about anywhere.
  - *Portable*: since it's interpreted, and highly standardized.
  - *High performance*: marketing bull. It's slow.
  - *Multithreaded*: you can have different threads running & communicating.
  - *Dynamic*: it's garbage collected; you can link in new code anytime.

Java -- "Simple & Object Oriented"

- It's like C in that...
  - Similar syntax for control (if, while, for) and basic assignment
  - Similar basic built in data types (int, float, etc)
  - You can make complex data types and allocate them as needed
- It's like C++ in that...
  - You can do object oriented design
  - You can declare classes of objects, the classes have methods attached, you can define an instance (allocate) a member of the class
  - You get all the usual encapsulation & abstraction benefits

Java -- Primitive Data Types

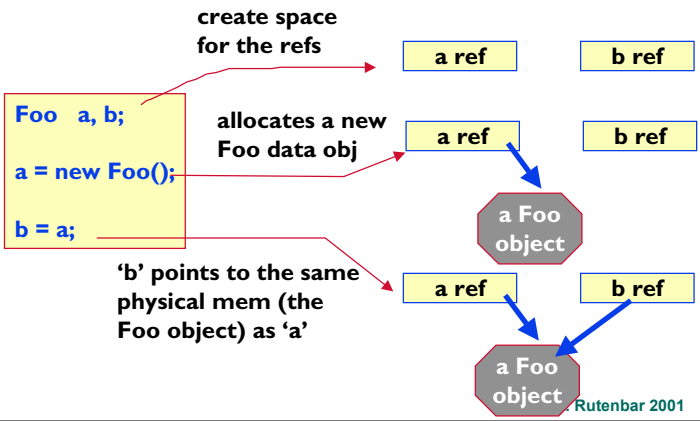
- Primitive means storage allocated just for this item
  - There are no pointers or reference parts to this object
  - Similar to C, not exactly the same

Type	Contains	Size	Comments
boolean	true, false	1 bit	explicit part of Java, not #define
char	Unicode character	16 bits	not 8! for international chars
byte	signed integer	8 bits	from -128 to +127 (no unsigned)
short	short integer	16 bits	from -32k to +32k
int	signed integer	32 bits	from -2B to +2B
long	signed integer	64 bits	it's *big*
float	IEEE std floating pt	32 bits	standard 32 bit real
double	IEEE std floating pt	64 bits	standard 64 bit real

- Aside: naming conventions
  - Primitive types all start with lower case letters
  - Other stuff (called "reference" types, start with capital letters)

Java -- Reference Types

- Everything that's not primitive is a *reference* type
  - This means objects that get declared and instantiated
  - It also means arrays
  - "Reference" here means like a pointer in ordinary C.



Java -- Arrays

- Basically like C++
  - In general: elementType[] varName = new elementType[arraySize]
  - You declare the ref...then you have to allocate the guts of the array

```
class Gauss {
public static void main(String[] args) {
      int[] ia = new int[101];
      for (int i = 0; i < ia.length; i++)
            ia[i] = i;
      int sum = 0;
      for (int i = 0; i < ia.length; i++)
            sum += ia[i];
      System.out.println(sum);
      }
}
```

Aside: About Object Oriented Programming

- When you took a data structures class..
  - They told you (I hope) 2 reasons we make complex data structures
  - **Abstraction:** hides the dirty details of the implementation of the data object you want to use. You have to do a bunch of pointer chasing and special case code to implement a STACK properly, but why show all this gruesome stuff to the world?
  - **Encapsulation:** you can put all your related data objects and the procedures that operate on them in one tidy little bundle. The outside world sees the object and the methods that work on the object, but not the gruesome details, which are hidden.
- In C++ and in JAVA...
  - Objects and methods are explicit parts of the language
  - But the philosophy and syntax are different

Java -- About Objects

A new kind of object is called a "class"

This class is named "Foo"

```
class Foo {

    variable define
    variable define
    ...

    method() {
    }

    method() {
    }

}
```

You define the *vars* which actually hold the data items for this objects here.

You define the *procedures* which actually operate on the data items for this object here. These are "methods"

To actually allocate a Foo object, do...

```
Foo aFooThing;
aFooThing = new Foo;
```

Declaring aFooThing is not same as allocating space for it

Simple & O-O: Why Java != C or C++

- There is no C pre-processor
  - You cannot do #define or #ifdef...#endif
  - Your code has to work *everyplace* you plan to run it w/o platform-specific modifications (unlike how people write UNIX code)
  - You cannot define any macros
- There are no #includes
  - You can import stuff from other Java files, but the mechanism is different, more like getting stuff from a library
- There are no global variables, no global procedures
  - Everything in Java is an object, operated on by object methods
  - Cannot just have naked global vars or functions floating around
  - Java enforces a very "pure" object-oriented programming model

Java versus C

Typical C file

```
#include  <file stuff>
...

#define  constants
#define macros(...)
...
#ifdef SOLARIS
   platform specific stuff
#endif

define global vars
define structs (typedefs)

routine_defns(..) {
}

main(char *argv[], int argc) {
   ...main code
}
```

!=

Typical Java file

```
import java libraries
...

class FOO {
   define vars
   method_defns(...) {
   }
}
class BAR {
   define vars
   method_defns(...) {
   }
}

class myStuff {
  public static void main(String[]) {
  ...main code
   }
}
```

Aside: Java Applications versus Applets

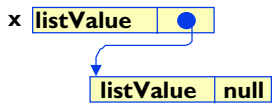
- Java application
  - A stand-alone program that will be run by the Java virtual machine on whatever platform you are sitting on
  - You compile the code, you get an executable, you run it
  - You can do pretty much anything you want in an application

- Java applet
  - A program that you intend to download from the network to your favorite browser, which will provide the Java virtual machine to actually run your code *inside the browser window*.
  - Applets must have a particular structure: there are certain methods they have to implement in certain ways in order for the browser to be able to run the code (eg, initialization code, drawing code, redraw code)
  - Strict security model: your browser sets how much the applet can do. Ex: Cannot necessarily open a file to write on your host machine unless you explicitly permit this. Cannot necessarily send out a packet on the network from a Java applet.

Simple & O-O: Why Java != C or C++

- There are no pointers. None. Zip. Nada.
  - No *foo stuff. No &foo stuff. No x = (Foo *)malloc(sizeof Foo) stuff.
  - No pointer arithmetic.
    - for(p = &array[0]; p!=NULL; p++) -- gone.
  - Objects have slots whose value is another object, accessed via "dot" notation.

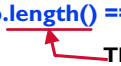
```
class ListNode {
   int listValue
   ListNode  nextNode;
   ...
   public addNode(int newValue) {
      // code to add a new node to list
   }
}

ListNode  x;
x = new ListNode;
x.nextNode = new ListNode
```



Simple & O-O: Why Java != C or C++

- Java is garbage collected
  - Get a new object via x = new Foo,
    (like C++, not like C, with x = (Foo *)malloc(sizeof Foo))
  - No C-style malloc() or free() or sizeof stuff
  - When an object is no longer referenced by some other object, the Java Virtual Machine collects it and returns it to the storage pool.
  - PRO: it's way easier to write code this way. Lots of bugs can't happen.
  - CON: it's slower code, less predictable, more overhead in time (when the garbage collector runs) and space (to tag the objects with the info necessary to collect them as needed)
- Strings are a real part of the language
  - "String" is a defined class, but compiler treats it specially
  - Example

```
String foo;
foo = "hello world";
if (foo.length() == 0)....
```



This is an *instance method*; more about this later...

Simple & O-O: Why Java != C or C++

- Basically same operator set (OK, so here Java ~ C)
  - Same basic arithmetic ops + - * / %, bit shift ops, logical compare ops, etc
- Small differences
  - + works on String objects: it concatenates them.
  - &, | work on integers to do parallel bit-wise ops, but they also work on booleans to do logical ops. These always evaluate *all* their operands on left and right side, even if value of expression is known after only partial processing of the expression.
  - A few others (see, eg, Java in a Nutshell.)
- Difference: no operator overloading
  - + is always "plus", can only do it on numbers and Strings. Period.
  - Cannot make it work on ComplexNum, Matrices, other defined classes.
  - Simplifies reading the language.

Simple & O-O: Why Java != C or C++

- Similar control structures.

- if() ... else...; while(); do/while();

  Pretty much the same EXCEPT the test MUST return a boolean

```
int i = 10;
while (i- -) {
   Object p = getObject();
   if (p) {
      do something;
   }
}
```

Illegal. In C it works since when i == 0 loop quits. But (int)0 != boolean false in Java language.

Illegal. In C it works since when p==NULL (==0) loop quits. But NULL != boolean false in Java language.

- Switch statement is the same.
  - You can use byte, short, int, long, char as the case statement labels.

Simple && O-O: Why Java != C or C++

- for( ) loops basically same
  - Can define the vars in the loop & initialize, like in C++

```
for( int i = 0;  String s="hello"; (i<10) && (s.length() != 0); i++) ...
```

Remember, this MUST be a boolean

- No goto at all. Labeled break & continue.
  - Labels are the target of the implicit "goto" of the break or continue

```
test:  if( check(i) ) {
   for( int j=0; j<10; j++) {
      if ( j > i ) break;
      if ( a[i][j] == null ) break test;
   }
}
...
```

Just break this loop

Breaks out of both loops down to here-- which is the end of the "test" block

Simple & O-O: Why Java != C or C++

- Exception handling is much more elegant
  - When something screws up the Java machine "throws an exception" which is just another kind of object, with values and methods
  - You can "catch" that exception and decide how to handle it.

```
try {
   // normally this code runs top to bottom, unless there is a problem
}
catch (SomeException  oops1) {
   //handle the exception object oops1 of type SomeException  here
}
catch (AnotherException oops2) {
   // handle different exception object oops2 of type AnotherException
}
finally {
   // always execute this code regardless of whether we leave the try{}
   // block normally, or via a handled exception, or via an unhandled
   // exception, or via a break, continue or return statement
}
```


Simple & O-O: Why Java != C or C++

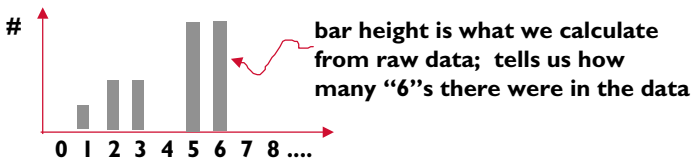
- So, Java can't (isn't supposed to) core dump
  - It's interpreted, so anything that screws up is caught by the Java virtual machine, and generates an exception object.
  - If you catch that exception, you can decide how to proceed.
  - If you don't catch it, the Java machine just stops executing your code and tells you where the problem happened
- Consequences
  - Cannot have a memory leak.
  - If you deref a null pointer -- Java throws an exception.
  - If you divide a number by 0 -- Java throws an exception.
  - If you walk off the end of an array -- Java throws an exception. etc etc.
  - *Pretty nice environment in which to debug code.*

Examples...

- You can go look up the rest of the syntax...
  - Find your favorite Java book.
  - The first 100 or so pages of Java in a Nutshell are pretty good here.
- Eventually, you want to go look at real code
  - Nice thing about Java is there is a LOT of Java code out there
  - Can go out on the net and get it and read it and run it
  - Applet code even runs in your browser.

Ex1: Histogram Class

- Task
  - You have numerical data and want to calculate a histogram on the data.



  - I want a class that implements an abstract histogram type on integers, and I want to be able to do 2 things:
    - Add a new raw data element to the histogram
    - Print out the histogram data, including mean/median

Ex1: Histogram Code (part 1)

```
class Histogram {

private final int  SIZE = 200;

public  int[ ] histArray = new int[SIZE];
```

2 vars, one (SIZE) is hidden;
the other (the actual histogram array)
is public

```
    public Histogram() {

        for( int i = 0; i<SIZE; i++)
            this.histArray[i] = 0;
    }
```

Constructor for the class;
when you say
Histogram H = new Histogram();
"new" calls this code

```
    public void add( int i ) {
        this.histArray[i]++;
    }
```

Simple method to add a data pt
to one element of histogram

...

Ex1: Histogram Code (part 2)

```
public void print(String title) {
        int tot = 0;
        int count = 0;
        int top = SIZE-1;
        int i;

        for( i=SIZE-1; i>=0; i--) {
            if( this.histArray[i] != 0 ) {
                top = i;
                break;
            }
        }

        System.out.println("\n-----------------------------------------------------");
        System.out.println(title);

        for(  i = 0; i<=top; i++) {
            System.out.println(i + "\t" + this.histArray[i]);
            tot +=  (i * this.histArray[i]);
            count += this.histArray[i] ;
        }
```

Ex1: Histogram Code (part 3)

```
        ...
        System.out.println("  mean = " + ( (double)tot / (double)count)  );

        int medianNum =  (int) Math.round ( ( (double) count / 2.0 ) );

        int first, last = 0;
        for(i=0; i<=top; i++ ) {
              if( this.histArray[i]  ==0 )
                    continue;

              first = last + 1;
              last = first + this.histArray[i] - 1;

              if( first <= medianNum && last >= medianNum) {
                    System.out.println("  median = " + i );
                    break;
              }
        }

    }
}
```


Using the Histogram Code

```
import java.lang.Math;

class doHist {

    public static void main(String args[]) {

        Histogram  aHist = new Histogram();

        for ( each piece of data    int x    I want to analyze )
              aHist.add(x);

        aHist.print("Num     Count");
    }

}
```

```
Num  Count
0        0
1        20
2        19
3        10
4        11
5        19
6        11
7        8
8        7
9        4
10       3
11       1
12       2
13       0
14       1
 mean = 4.48276
 median = 4
```

Example of output you get on "stdout"

Issues to Address

- Several things brought up by that example
- Object definition
  - Is this "class name { }" thing all there is?
- Computation
  - Does every real calculation" look like objectName.method(params)?
- Input Output
  - We saw the Java version of C's printf: System.out.println("string foo");
  - What else is there? How do I open a file and read it? Write it?
  - How do I do simple graphics, eg, open a window, draw a rectangle?

Java Objects: Class Definitions

- There is subtlety here we should be clear on.
- Example: a "circle" class (from *Java in a Nutshell*)

```
public class Circle {
   static int numCircles = 0;  // class variable, count num of instances
   public double x, y, r;      // instance variables: center and radius

   public Circle(double x, double y, double r) {
       this.x = x;  this.y=y;  this.r = r;
       numCircles++;
   }

   public Circle(double r) { this(0.0, 0.0, r); }

   public Circle biggerInst(Circle c) {
       if (c.r > r) return c;  else return this;
   }

   public static Circle biggerClass(Circle a, Circle b) {
       if(a.r > b.r) return a;  else return b;
   }
}
```

Constructor takes 3 nums

Constructor just takes radius, sets other to 0

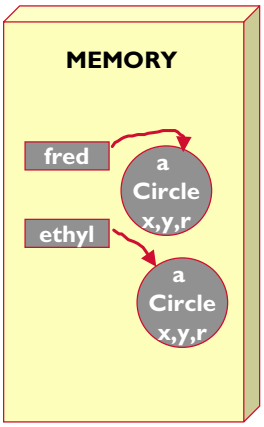
2 methods...

Class Defn: Instance Variables

- You get a fresh copy with *every* new instance of object

```
public class Circle {
   static int numCircles = 0;  // class variable, count num of instances
   public double x, y, r;   // instance variables: center and radius

   public Circle(double x, double y, double r) {
        this.x = x;  this.y=y;  this.r = r;
        numCircles++;
    }
    public Circle(double r) { this(0.0, 0.0, r); }

    public Circle biggerInst(Circle c) {
        if (c.r > r) return c;  else return this;
    }
    public static Circle biggerClass(Circle a, Circle b) {
        if(a.r > b.r) return a;  else return b;
    }
}
```

```
Circle fred, ethyl;
fred = new Circle(0.0,2.0,3.0);
ethyl = new Circle(5.0);
```

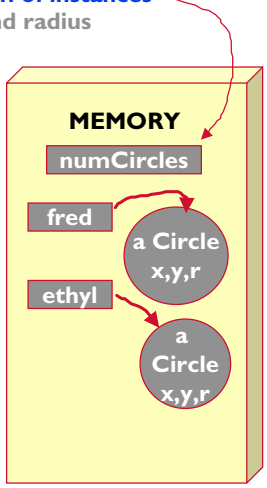


Class Defn: Class Variables

- You get *one* copy only; belongs to the class itself

```
public class Circle {
   static int numCircles = 0;  // class variable, count num of instances
   public double x, y, r;   // instance variables: center and radius

   public Circle(double x, double y, double r) {
        this.x = x;  this.y=y;  this.r = r;
        numCircles++;
    }
    public Circle(double r) { this(0.0, 0.0, r); }

    public Circle biggerInst(Circle c) {
        if (c.r > r) return c;  else return this;
    }
    public static Circle biggerClass(Circle a, Circle b) {
        if(a.r > b.r) return a;  else return b;
    }
}
```

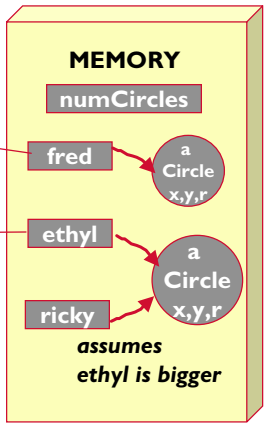
```
Circle fred, ethyl;
fred = new Circle(0.0,2.0,3.0);
ethyl = new Circle(5.0);
```



Class Defn: Instance Method

- Routine that works on a particular *instance* of a Circle

```
public class Circle {
   static int numCircles = 0;  // class variable, count num of instances
   public double x, y, r;   // instance variables: center and radius

   public Circle(double x, double y, double r) {
        this.x = x;  this.y=y;  this.r = r;
        numCircles++;
    }
    public Circle(double r) { this(0.0, 0.0, r); }

    public Circle biggerInst(Circle c) {
        if (c.r > r) return c;  else return this;
    }
    public static Circle biggerClass(Circle a, Circle b) {
        if(a.r > b.r) return a;  else return b;
    }
}
```



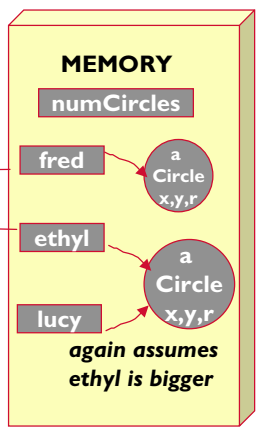

```
Circle ricky = ethyl.biggerInst(fred);
```

a particular instance of a Circle — instance method — a parameter that is a Circle

Class Defn: Class Method

- Routine that belongs to *class* Circle, *not* a specific inst.

```
public class Circle {
   static int numCircles = 0;  // class variable, count num of instances
   public double x, y, r;   // instance variables: center and radius

   public Circle(double x, double y, double r) {
        this.x = x;  this.y=y;  this.r = r;
        numCircles++;
    }
    public Circle(double r) { this(0.0, 0.0, r); }

    public Circle biggerInst(Circle c) {
        if (c.r > r) return c;  else return this;
    }
    public static Circle biggerClass(Circle a, Circle b) {
        if(a.r > b.r) return a;  else return b;
    }
}
```



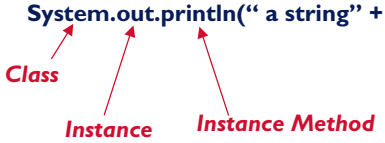

```
Circle lucy = biggerClass(fred, ethyl);
```

Class method — 2 parameters that are Circles

Why This Matters

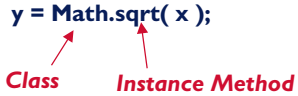
- It explains a *whole lot* of arcane Java syntax
- Example: how do you print to stdout?

```
System.out.println(" a string" + var + "another string");
```



- Example: how do you do a square root?

```
double  y = Math.sqrt( x );
```



Why This Matters

- How do you compare 2 strings?
- *Not* like this

```
String s = "hello";
if ( s == "hello") ...
```

- Like this

```
String s = "hello";
if ( s.equals("hello") )
```

Class Defn: Class Constructors

- Routine(s) with same name as class, called by "new"

```
public class Circle {
   static int numCircles = 0;  // class variable, count num of instances
   public double x, y, r;   // instance variables: center and radius

   public Circle(double x, double y, double r) {
        this.x = x;  this.y=y;  this.r = r;
        numCircles++;
    }
    public Circle(double r) { this(0.0, 0.0, r); }

    public Circle biggerInst(Circle c) {
        if (c.r > r) return c;  else return this;
    }
    public static Circle biggerClass(Circle a, Circle b) {
        if(a.r > b.r) return a;  else return b;
    }
}
Circle fred, ethyl;
fred = new Circle(0.0,2.0,3.0);
ethyl = new Circle(5.0);
```


Class Defn: Class Constructors

- Can have different versions that take different args (like C++)

```
public class Circle {
   static int numCircles = 0;  // class variable, count num of instances
   public double x, y, r;   // instance variables: center and radius

   public Circle(double x, double y, double r) {
        this.x = x;  this.y=y;  this.r = r;
        numCircles++;
    }
    public Circle(double r) { this(0.0, 0.0, r); }

    public Circle biggerInst(Circle c) {
        if (c.r > r) return c;  else return this;
    }
    public static Circle biggerClass(Circle a, Circle b) {
        if(a.r > b.r) return a;  else return b;
    }
}
Circle fred, ethyl;
fred = new Circle(0.0,2.0,3.0);
ethyl = new Circle(5.0);
```

Class Defn: Class Destructor

- In Java it's called a "finalizer"
  - It's an instance method (non-static), takes no args, returns void, must be called *finalize*()
- What finalize() does NOT do
  - It doesn't delete your object memory.
  - Java is garbage collected so when something has no users referencing it, it gets automatically deleted
  - Unlike C++, where you'd need a ~Circle() method to kill the object
- What finalize() does do
  - It releases resources that the garbage collector cannot see
  - Examples: file descriptors, network sockets, etc.
  - Stuff you want to "close" before you quit
  - Java calls the finalizer() before it garbage collects the object

Class Defns: Inheritance

```
class Circle  {
...
}
```

inherits....

```
class GraphicsCircle  {
...
}
```

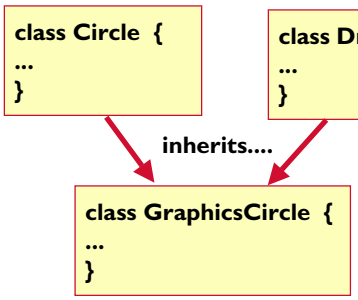
Want a new kind of circle that has all the old properties, but also has new methods to draw a circle on the screen.

Don't have to make a whole new class, you inherit the old stuff

```
public class GraphicCircle extends Circle {
   // automatically inherit old Circle
   // methods and vars
   // we omit a constructor for clarity

   Color  outline, fill;
   public void draw(DrawWindow dw) {
       dw.drawCircle(x, y, r, outline, fill);
   }
}

//use it -- examples
GraphicCircle  c = new GraphicCircle;
GraphicCircle  d = new GraphicCircle;
c.draw( aWindow );
Circle e = biggerClass(c, d);
```

Class Defns: Multiple Inheritance

- Can I inherit from > 1 super-class? Basically, no....



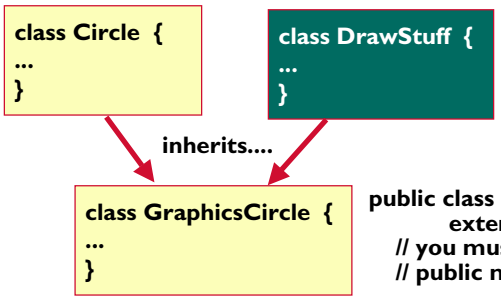
In C++ you can inherit the actual implementation of different methods from more than one parent class.

In Java, you can inherit actual method implementations from just 1 parent class.

So -- is there something *like* multiple inheritance in Java? yes...

Class Defns: Interfaces

- An interface is just a class def that defines the *names* and *params* of its methods, but no "code guts" for them.




```
public interface DrawStuff {
   public void aMethod(.....);
   public void bMethod(.....);
}
```

```
public class GraphicCircle
        extends Circle implements DrawStuff {
   // you must add real code for all the
   // public methods in Draw Stuff

    public void aMethod(...) {
       code;  code;   code;
    }
    public void bMethod(...) {
       code;  code;   code;
    }
}
```

Aside: So, Now You Know How to do an Applet

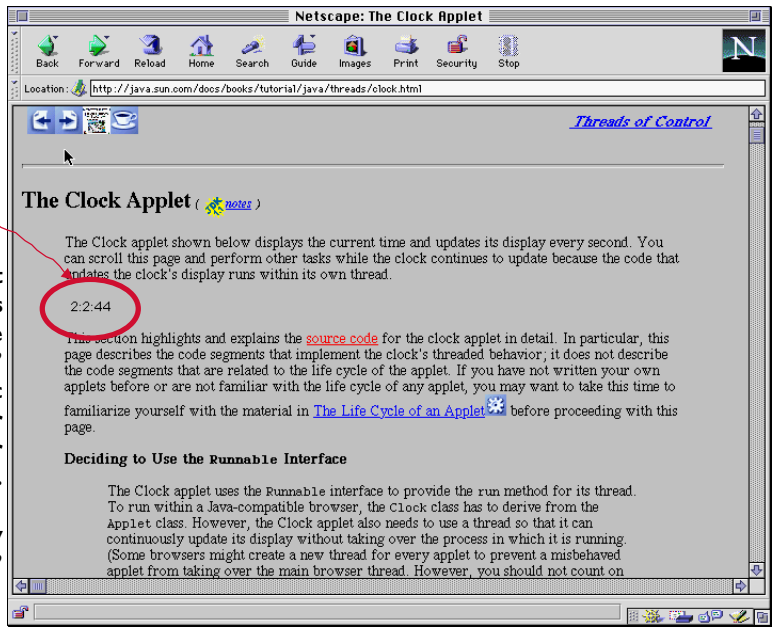
```
import java.awt.Graphics;
import java.util.Date;

public class Clock extends java.applet.Applet implements Runnable {

   Thread clockThread = null;

   public void start() {
      if (clockThread == null) {
         clockThread = new Thread(this, "Clock");
         clockThread.start();
      }
   }
   public void run() {
      // loop terminates when clockThread is set to null in stop()
      while (Thread.currentThread() == clockThread) {
         repaint();
         try {
            Thread.sleep(1000);
         } catch (InterruptedException e){
         }
      }
   }
   public void paint(Graphics g) {
      Date now = new Date();
      g.drawString(now.getHours() + ":" + now.getMinutes() + ":" + now.getSeconds(), 5, 10);
   }
   public void stop() {
      clockThread = null;
   }
}
```

We must implement these methods (which are from the Runnable interface) so a browser knows what to call to "run" our applet

Clock Applet, Running

This applet just updates a little clock "text" every sec in your browser window.

Pretty cool, huh?



Java I/O

- Pretty rich set of features
  - Support for C-like command line input
  - Support for networks
  - Support for graphics, and interacting with a browser
  - Support for file (byte stream) IO from your local host file system
- We will look at just 3 of these, a little bit
  - Doing command line input
  - Doing file stream IO
  - Very simple paint-a-rectangle-on-a-window sort of graphics

Command Line I/O

- A lot like ordinary C, but with nicer string vars
  - Also platform dependent. You may actually type on a command line to run the code + its arguments, or Java may pop up a window and ask.

```
public static void main (String args[]) {
    String Infile = new String("example.in");
    String Outfile = new String("example.out");
    if (args.length == 1) {
      Infile = args[0];
    }
    if (args.length == 2) {
      Infile = args[0];
      Outfile = args[1];
    }
    if (args.length > 2) {
      System.err.println("I don't know how to handle that many command " +
                                    "line arguments.");
      System.err.println("Please try again.\n");
      System.exit(-1);
    }
}
```

File I/O: Opening and Closing the FIile

- Java has rich support for various kinds of *data streams*

```
try {
      DataInputStream Inpfile =
            new DataInputStream(new FileInputStream(Infile));
      String LineIn = Inpfile.readLine();
      if (LineIn == null) {
            System.err.println("There is no data on the first line of the " +
                        "input file.");
            System.err.println("Not cool, get a real data file.");
            System.exit(-1);
      }
     ... // stuff to read the file...
      Inpfile.close();
} catch (IOException e) {
       System.err.println("\nWe are having trouble opening and reading " +
             "the file: " + Infile);
       System.err.println("We get the error: " + e + "\n");
       System.exit(-1);
}
```


File I/O: Actually Reading the FIle

- How do I do fscanf(fileID, "%d %d", &X, &Y)...?
  - Several ways involving a *tokenizer*
  - You *tokenize* a file stream by recognizing that the chars really form *tokens* that represent readable stuff like ints and floats and Strings, which are separated by "white space" like spaces and tabs.
  - A *tokenizer* skips over the white space and hands you the next token.
  - If you know what to expect, you can just parse for the right int, float, etc
  - (If not (more general case) you can ask Java what the next token was.)

```
String LineIn = Inpfile.readLine();
...
StringTokenizer DataIn = new StringTokenizer(LineIn, " ");
X = Integer.parseInt(DataIn.nextToken());
Y = Integer.parseInt(DataIn.nextToken());
Width = Integer.parseInt(DataIn.nextToken());
Height = Integer.parseInt(DataIn.nextToken());
LineIn = Inpfile.readLine();
DataIn = new StringTokenizer(LineIn, " ");
R = Integer.parseInt(DataIn.nextToken());
G = Integer.parseInt(DataIn.nextToken());
B = Integer.parseInt(DataIn.nextToken());
```

String to read

White-space defn

File I/O: Subtle Stuff

- String LineIn = Inpfile.readLine();
  - Reads one return-delimited line from Inpfile and sticks it in a String
- StringTokenizer DataIn = new StringTokenizer(LineIn, " ");
  - Looks at the String LineIn, and tells the tokenizer that whitespace = " "
  - Makes a new "tokenized" object that we can yank tokens (inputs) out of, one by one, in order
- X = Integer.parseInt(DataIn.nextToken());
  - We expect this to be a fscanf(file, "%d", &x) sort of thing...
  - X is just an int, ie, int X;
  - But "int" is not an object, it's a primitive type
  - All primitive types have an associated class object def--in this case Integer-- where useful class methods live.
  - The useful method we want is "parseInt" which yanks the next int out of this tokenized data stream, and returns a primitive int

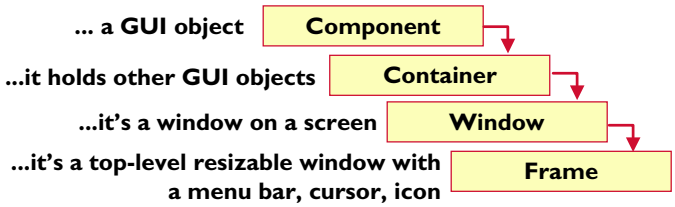
File I/O: Printing to a FIle

- Very similar stream-based idea
  - Remember how ordinary stdout was: System.out.println("stuff")...?

```
try {
        FileOutputStream ostream = new FileOutputStream(Outfile);
        PrintStream Ourprint = new PrintStream(ostream);
        Ourprint.println("Here is the results of our 760 Java example " +
                                        "program.");
        Ourprint.println("The data on the rectangle we drew is:\n\t" +
                                        ADrawnRect + "\n");
        Ourprint.close();
} catch (IOException e) {
        System.err.println("\nWe are having trouble writing to the file: " +
                                        Outfile);
        System.err.println("We get the error: " + e + "\n");
        System.exit(-1);
}
```

Java I/O: Simple Drawing

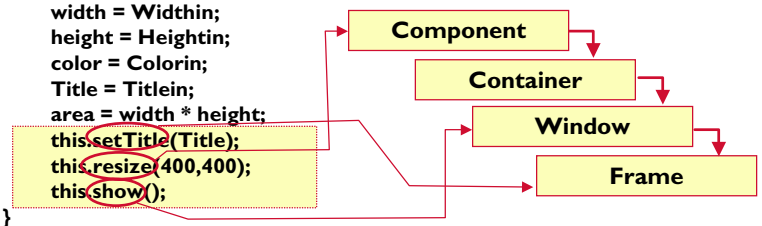
- Pretty rich set of drawing classes called "awt"
  - Java "abstract windowing toolkit"; get it via: import java.awt.*;
- 3 kinds of objects in the awt
  - Graphics: classes that define colors, fonts, images, etc
  - Components: classes are graphical user interface (GUI) components like buttons, menus, lists, dialog boxes
  - Layout managers: classes that control the layout of other graphics components within their Java "containers"
- Java hierarchy for us



Example: Drawing a Rectangle in a Frame

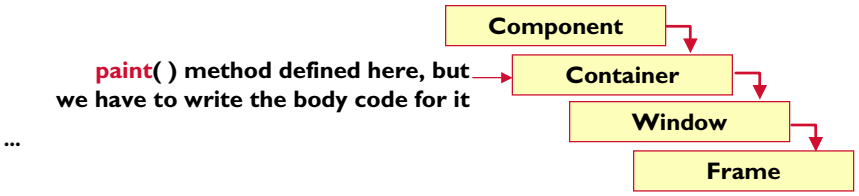
- From our small example again

```
class my_rect extends Frame {
        private int x,y;
        private int height, width;
        private int area;
        private Color color;
        private String Title;

        //constructor
        public my_rect(int Xin, int Yin, int Widthin, int Heightin, Color Colorin,
              String Titlein) {
            x = Xin;
            y = Yin;
            width = Widthin;
            height = Heightin;
            color = Colorin;
            Title = Titlein;
            area = width * height;
            this.setTitle(Title);
            this.resize(400,400);
            this.show();
        }
```



lines show where we inherit these methods

Example: Drawing a Rectangle in a Frame



paint( ) method defined here, but we have to write the body code for it

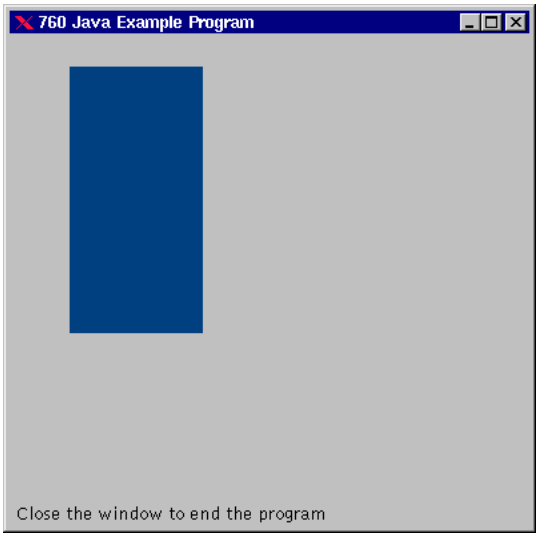
...

```
/* This paint procedure is called by the system, to update the window.
   All drawing objects need to live here. */
public void paint(Graphics g) {
      g.setColor(color);
      g.fillRect(x,y,width,height);
      g.setColor(Color.black);
      g.drawString("Close the window to end the program",10,390);
}
```

The Graphics class defines the platform-dependent drawing primitive. A "graphics" object gets allocated by Java (by Container) and passed to the paint( ) method. Then you call graphics instance methods to really draw stuff

What Does This Code Actually Do...?

- To run it:
  - Put code in "example.java" and data in "example.in"
  - To compile, type: javac example.java
  - Generates ".class" files holding Java byte codes for all your classes
  - To run it, type: java example
  - It does this when run on IBM AIX UNIX...



Aside: AWT versus Swing in JAVA

- Newer graphics components called "Swing"
  - Quoting from http://pandonia.canberra.edu.au/java/swingtut/tut2.html:
- Evolution of Java GUI programming
  - The AWT (Abstract Window Toolkit) has been present in all versions of Java
  - The AWT objects are built above native code objects, giving a native look-and-feel
  - The AWT objects are a least common denominator of all platforms
  - The Swing objects are a separate library for [old, early] JDK 1.1
  - The Swing objects are in pure Java, and have the same look-and-feel on all platforms
  - The L&F [look & feel] of Swing objects can be customised to particular styles
  - In JDK 1.2, the Swing objects are part of the Java Foundation Classes
  - The JFC objects will provide a superset of each platform's objects
  - The AWT objects will decrease in importance over time

Java Summary

- Interesting, elegantly designed O-O language
  - C-like syntax, but O-O stuff much tidier than in C++
  - Rich collection of libraries and reusable objects
  - Fairly portable (all the UNIXs, Windows, Mac, etc)
  - Interpreted & garbage collected & strongly typed: pretty robust
- Do people really do CAD in this?
  - Well—not much. People are doing GUIs and some configurable IP.
  - People are doing network downloadable applets now.
  - But "real" applications? -- Java is still too slow and too much memory
  - Compiler guys are working on real (not interpreted, fast) versions
- Why for 760?
  - It's very portable. Has nice library of useful data structures, methods.
  - As a nice, friendly prototyping language, it's hard to beat.

…But, There is Some JAVA CAD Out There…

August 31 2001
EETimes online
See:
http://www.eetimes.com/story/OEG20010831S0086