SACRAMENTO
STATE

Computer Science 133

Object-Oriented Computer Graphics Programming

Table Of Contents

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

I - Course Overview

- classroom conduct
- prerequisites
- course topics
- texts and references
- grading: exams and programs
- communication
- workload
- ethics

CSc-133 Lecture Notes

CSc 133 – Object-Oriented Computer Graphics Programming

Computer Science Dept.
CSUS

Contacting Your Instructor

Dr. John Clevenger (Sections 1 & 2)
Office: Riverside Hall 5018
Phone: (916) 278-6087
Office Hours: T.B.A.
Email: clevenger@csus.edu
Webpage: http://ecs.csus.edu/~clevengr

Dr. Pinar Muyan-Ozcelik (Section 3)
Office: Riverside Hall 5008
Phone: (916) 278-6713
Office Hours: T.B.A.
Email: pmuyan@ecs.csus.edu
Webpage: http://ecs.csus.edu/~pmuyan

Classroom Etiquette

This course requires concentration and focus!

Out of respect for others in the room:

Cell Phones:	off
Beepers:	off
Pagers:	off

and please refrain from:

browsing, facebooking, social networking, texting, instant messaging, tweeting, blogging, gaming, eBaying, day-trading, etc. etc., during class!

Prerequisites

- CSc 130 (Algorithms and Data Structures)
- CSc 131 (Intro. to Software Engineering)

… which implies:

- CSc 15 (Programming Methodology I)
- CSc 20 (Programming Methodology II)
- CSc 28 (Discrete Structures)
- Math 29 (Pre-calculus Math)

Prerequisites By Topic

Programming Experience

- 3 semesters in Java, C++, Ada, or similar OOP.
- Object-based principles: class/object definitions, method invocation, public vs. private fields, etc.
- Algorithms/data structures: lists, stacks, trees, hashtables, recursion

Software Engineering Topics

- Life Cycle: requirements, design, implementation, testing
- UML: Class, use-case, sequence diagrams

Math Topics

- Polynomial equations, trigonometric functions, matrix operations
- Cartesian coordinates, vectors, coordinate transformations

Prerequisite Verification

- Submit MySacState “Unofficial Transcript”
  - Deadline: <u>TUESDAY</u> of <u>WEEK TWO</u>
- Highlight your prerequisite(s)
- Yes, you really NEED the prerequisites!
- Transfer students: submit equivalent
- People with a “story”: see your instructor…

Repeat Policy

- Repeating a course *for the third time* (i.e., taking it for a *fourth* – or greater – time) requires filing a *Repeat Petition*
  - Available at the CSc Dept. Office (RVR 3018) or at *http://www.ecs.csus.edu/wcm/csc/forms.html*
  - Requires *Instructor, Dept. Chair,* and *Dean's* signature
- Deadline: TUESDAY of WEEK TWO

What is this course about ?

Two main topics:

1. Fundamentals of the "O-O" paradigm
2. Introduction to Computer Graphics

First topic: **Object-Oriented Paradigm**

- Language implementation:
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism
- Tools supporting OOA/OOD/OOP:
  - formalisms such as *UML*
  - *design patterns*
  - *application frameworks*

Second topic: **computer graphics**

- User interface (“GUI”) mechanisms
- Event-driven programming
- Basic line and polygon drawing
- Object, World, Screen coordinate systems
- Geometric transformations
- Devices and color models
- Basic animation

Texts and References

- Required Texts:
  - CSc 133 Lecture Notes,
    Clevenger/Gordon, Hornet Bookstore
- Recommended Texts:
  - Object-Oriented Design & Patterns, 2nd Ed.,
    Cay Horstmann, John Wiley & Sons,
    ISBN 0-471-74487-5
  - Schaum’s Outlines: Computer Graphics, 2nd Ed.,
    Xiang and Plastock, McGraw-Hill,
    ISBN 0-07-135781-5

Grading

- Weighted Curve based on:

Programming Assignments (4)	~40%
Midterm Exam	~25%
Final Exam	~30%
Attendance, Quizzes, etc.	~ 5%

- Additional Criteria
  - Satisfactory completion of:
    - Programming assignments (combined)
    - Exams (Midterm + Final combined)

Grading (cont.)

Programming Assignments

- there will be four (4) programming assignments
- *individual* work
- not necessarily equally-weighted
- They will be *cumulative!* Don’t try to skip one!
- Requirements:
  - Java (1.7+)
  - IDE: optional
- Late penalty: 4% per day up to 50%
- Must keep a *backup* (machine-readable) copy

Grading (cont.)

Exams

- Dates are noted on the schedule
- Final Exam as scheduled by University
- Study Guides will be provided
  - *but only the course notes are complete!*
- Make-up exams only under extreme circumstances:
  - *generally requires prior arrangements*

Communication

- CSc133 Mailing List
- SacCT:
  - assignments
  - announcements
  - feedback and grades
- E-mail your instructor your name / email address (by Tuesday of week 2)
- Check your email and SacCT daily

Workload

- “Freshman Counseling”:
  - 1 unit = 1 hr/wk in class + 2-3 hrs/wk outside, *on average, University-wide*
  - 3 units = 9-12 hrs/wk, *on average, University-wide*
  - 12 units = 36-48 hrs/wk, *on average, University-wide*
- Not all classes are “average”!
- This is a programming-intensive course

Ethics

- Submitting work *constitutes an agreement* that *the work is solely your own*
- Students who violate the University policy on academic honesty are:
  - **Automatically Failed**
  - **Referred to the Dean of Students**
- Detailed Ethics policies given in syllabus and posted on CSc-133 webpage

Reminder

By Tuesday of week 2:

- turn in a copy of your transcript, with prerequisites highlighted
- turn in “Academic Integrity” form
- register on “csc133” mailing list
- email your name and preferred email address to your instructor

<< This page intentionally left (almost) blank >>

II – About Java

- Compiling and Executing
- Java Syntax and Types
- Classes, instantiation, constructors, overloading
- References, Strings
- Garbage Collection
- Arrays
- Dynamic Array Types, Vectors, ArrayLists
- Parameter Passing
- Differences between Java and C++

SAMPLE PROGRAM:

- Save the following code in file “HelloWorld.java”:

```
import java.lang.*;
public class HelloWorld {
     public static void main ( String [ ] args ) {
          Greeter newGreeter = new Greeter();
          newGreeter.sayHello();
     }
}
```

- Save the following code in file “Greeter.java”

```
public class Greeter{
     public void sayHello()   {
          System.out.println (“Hello World!”) ;
     }
}
```

- Compile using the command

```
javac   HelloWorld.java
```

(implicitly also compiles Greeter.java)

- Execute HelloWorld using the command

```
java  HelloWorld
```

COMPILING:

```
% javac   MyProg.java              // tells the Java Compiler to compile the Java
                                   //  source code in the file named “MyProg.java”;
                                   //  file “MyProg.java ” must contain a ‘class’
                                   //  whose name is “MyProg”;
                                   //  produces an output “bytecode” file named
                                   //  “MyProg.class”
```

EXECUTING:

```
% java   MyProg                    // runs the JVM (“Java Virtual Machine”),  tells
                                   // it to “interpret”  (execute)
                                   //  the byte code in file “MyProg.class”
```

- File and Class names are case-sensitive (even in non-sensitive OS environments such as Windows)
- Source program file names *must* end in “.java”

ENVIRONMENT:

- Path to Java tools (“java”, “javac”) must be added to the “PATH” variable; e.g. in DOS:

```
set PATH=c:\jdk1.6\bin;%PATH%
```

- Path to Java libraries must be included in the “CLASSPATH” variable; e.g. in DOS:

```
set CLASSPATH=c:\jdk1.6\lib;%CLASSPATH%
```

Java Built-in Primitives (8 kinds):

INTEGER types:

- **byte** (-128 .. +127)
- **short** (2 bytes: -32768 .. +32767)
- **int** (4 bytes: -2G .. +2G)
- **long** (8 bytes: ± 2^{64})

REAL types:

- **float** (4 bytes, IEEE std., values denoted like “1.0F”)
- **double** (8 bytes, IEEE std., values denoted like “1.0”)

Additional types:

- **boolean** (*true* or *false)*
- **char** (16-bit “Unicode”)

Variable Declarations (primitives):

```
int  a ;

long  j = 4271843569L ;

double  x = 1.378 ;

char  c1 = ‘a’,   c2 = ‘z’ ;


int  i = 2 ;
int  k = i + 3 ;
```

Strong Typing:

```
float  x = 1.0 ;     // fails!   Must be 1.0F   (use double)

int j = 1 + ‘z’ ;     // fails!  Cannot mix types (unless “casting” is used)
```

MODIFIERS:

- Used to specify access and/or usage of classes, fields, and/or methods

- Visibility Modifiers
  - **`public`** // "world accessible"
  - **`private`** // "only accessible by methods in this class"
  - **`protected`** // "accessible by all classes in this 'package',
    // and all subclasses in *any* package"
  - <default> // "accessible by any class in this package"

- Additional Modifiers
  - **`static`** //"one for the whole class"
  - **`final`** //"restricted use" – e.g. variable cannot be changed
  - others to be seen later…

```
/** This is a sample class whose purpose is simply to show the form of several Java constructs. The class
 * provides a method which computes and prints the sum of all Odd integers between 1 and a  fixed Max
 *which do not lie inside a specified "cutoff range", and also are either divisible by 3 but  are not certain
 * "special rejected" numbers, or else are divisible by 5.   It does the calculation three times, expanding the
 * cutoff range each time.  It is assumed the Calculator is instantiated, and findSum is invoked, elsewhere.
  */
public class Calculator {
    public void findSum ()  {
        final int MAX = 100 ;
        int loopCount = 0 ;
        int lowerCutoff = 50;
        int upperCutoff = lowerCutoff + 25;
        int rangeExpansion = 10 ;

        while (loopCount < 3) {
            int sum = 0 ;
            for (int i=1; i<=MAX; i++) {
                //check for odd number outside cutoff range
                if ( (i%2 != 0) && ((i<lowerCutoff) || (i>upperCutoff)) ) {
                    //found an acceptable odd number; check if divisible by 3
                    if ((i%3 == 0))  {
                        switch (i)   {  //divisible by 3; check for special reject numbers
                           case 15: {
                               System.out.println ("Found and rejected 15");
                               break;
                           }
                           case 21:
                           case 27: {
                               System.out.println ("Found and rejected " + i);
                               break;
                           }
                           default: { sum = sum + i ; }
                        }
                    }  else {
                        //not divisible by 3; check if multiple of 5
                        if (i%5 == 0)
                           sum = sum + i ;
                    }
                }
            }
            System.out.println ("Loop " + loopCount + ":  sum = " + sum );
            lowerCutoff -= rangeExpansion; //increase the cutoff range
            upperCutoff += rangeExpansion;
            loopCount++;
        }
    }
}
```

CLASSES:

- Nearly every data item in a Java program is an **OBJECT**
    - (primitives are the exception)
- An *object* is an **INSTANCE** of a **CLASS**
    - Programmer-defined class, or
    - Class from a predefined library
- All code in a Java program is inside some class – even the *main* program
- Classes contain **FIELDS** and **METHODS** (also called *procedures* or *functions*)

```
public class  BankAccount {
    private double currentBalance ;

    private String ownerName = "Rufus" ;

    public int branchID = 405 ;


    public double getBalance() {
        return currentBalance ;
    }

    public void deposit (float amount){
        currentBalance += amount ;
    }
}
```

INSTANTIATION:

- Primitives (int, char, boolean, etc.) are not objects (in the OO/Java sense)
    - allocated as "local variables" on the stack

- Code in a class (e.g. the class containing the main program) can create objects by **INSTANTIATION**
    - Objects are allocated on the Dynamic Heap

- Example instantiations:

```
//assume the following user-defined class:

   public class Ball {

        private int xCenter, yCenter, radius;

        private Color ballColor ;

        //other fields here . . .

        //method declarations here . . .

   }


 //the following statements create INSTANCES of the Ball class:

   Ball  myBall  = new Ball();

   Ball  yourBall = new Ball();

 //the following statement creates an (initialized) INSTANCE
 //of the predefined Java class String:

   String myName = new String ("Rufus T. Whizbang");
```

CONSTRUCTORS:

- **Instantiation** is done using the **new** operator to invoke a "**CONSTRUCTOR**"

  - **No "implicit instantiation" like in C++**

    `Ball myBall;` **//** **creates** a Ball object in C++, but not in Java!

- Constructors always have exactly the same name (including case) as the CLASS

- Task of a constructor: **Create** and **Initialize** an object (an instance of the class)

- Programmer can define multiple constructors with different parameters (arguments)

- If the programmer provides NO constructors for a class, Java automatically provides a 'default' constructor with no parameters ("default no-arg constructor")

- NOTE: no "destructor" in Java [C++ "**~**" ; C "**free**" ; Pascal "**dispose**"]

  - **Objects are automatically "freed" when no longer accessible – handled via "garbage collection"**

CONSTRUCTOR EXAMPLES:

```
//assume the following user-defined class:
  import java.awt.Color;
  public class Ball {
        private int xCenter, yCenter, radius;
        private Color ballColor ;

        // (programmer-provided) no-arg constructor
        public Ball () {
              xCenter = yCenter = 0;
              radius = 1;
              ballColor = Color.red;
        }

        // constructor allowing specification of Color
        public Ball(Color theColor) {
              ballColor = theColor ;
              xCenter = yCenter = 0;
              radius = 1;
        }

        //methods (functions) provided by “Ball” objects
        public int getDiameter() {
              int diameter = 2 * radius ;
              return diameter ;
        }
  }


// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
//then the following would be typical instantiations appearing
//  in code in some program (class):
  . . .
  Ball myBall = new Ball();                 //a red ball of radius 1 at (0,0)

  Ball yourBall = new Ball(Color.blue);     //a blue ball, radius=1 at (0,0);
  . . .


  //invocations of methods in different objects (instances):
  int myDiameter = myBall.getDiameter();
  int yourDiameter = yourBall.getDiameter();
```

Overloading Constructors and Methods:

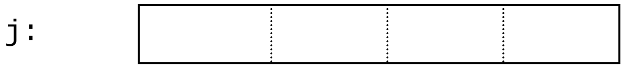
(example from Jia: OO Software Development, A-W, 2000)

```
/** This class gives a representation of a point,
  * showing examples of overloading.
  */
public class Point {
      private double x,y ;    //the coordinates of the point
      //overloaded constructors:
      public Point ()   {
            x = 0.0 ;
            y = 0.0 ;
      }
      public Point (double xVal, double yVal)   {
            x = xVal ;
            y = yVal ;
      }
      //overloaded methods:
      /** Returns the distance between this point and the other point */
      public double distance (Point otherPoint) {
            double dx = x - otherPoint.x ;
            double dy = y - otherPoint.y ;
            return Math.sqrt (dx*dx + dy*dy) ;
      }
      /** Returns the distance between this point and a location */
      public double distance (double xVal, double yVal)     {
            double dx = x - xVal ;
            double dy = y - yVal ;
            return Math.sqrt (dx*dx + dy*dy) ;
      }
      /** Returns the distance between this point and the origin */
      public double distance ()     {
            return Math.sqrt (x*x + y*y) ;
      }
}
```

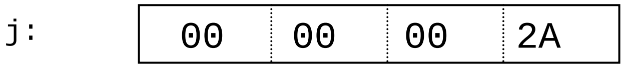
REFERENCES:

- A variable of a primitive type holds a data value of that type:

```
int  j ;        //declaration allocates space for j:
```

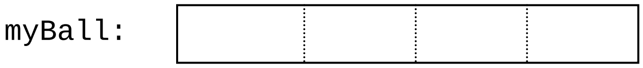


```
j = 42 ;        //assignment stores a value in the space:
```



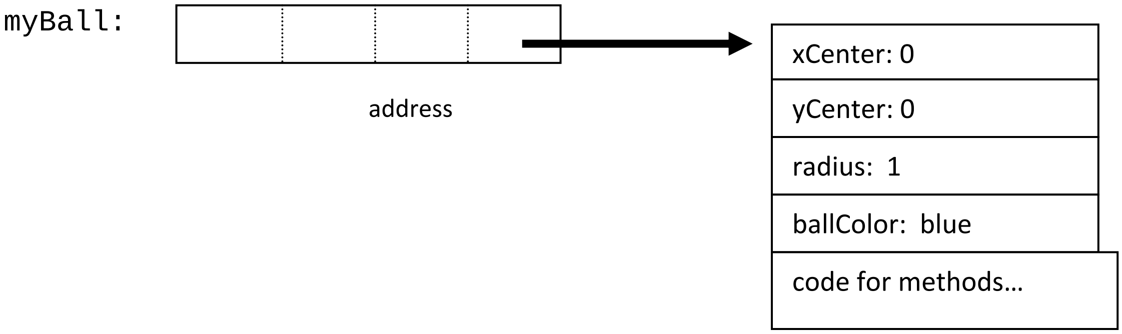
- A variable representing an object holds the address of the object, called a *reference*:

```
Ball myBall ;   //declaration allocates space for a pointer:
```



- Construction of an object allocates space on the Heap for the object, and sets the reference to point to the object:

```
myBall = new Ball(Color.blue);   // create the object:
```



REFERENCES, cont. :

- All object values (fields and methods) are accessed via a reference:

```
Ball myBall = new Ball(Color.blue);

Color myColor = myBall.ballColor;     //access object color

int diameter = myBall.getDiameter();  //invoke object
method
```

- A “reference” is essentially a pointer that doesn’t have to be “dereferenced”:

  Pascal-like dereference :

```
myBall : pointer to Ball ;
myColor :=  myBall^ballColor;
```

  C-like dereference :

```
Ball * myBall ;
myColor =  myBall -> ballColor;
```

- C++ also has references (most programmers don’t use them)
- Java ***only*** has references – no other way to access objects
- Java does not allow “pointer (reference) arithmetic”

REFERENCES vs. PRIMITIVES:

- Consider the following code which uses primitives:

```
...
int a = 42;
int b = a;
System.out.println (a);         //prints 42
b = 3;
System.out.println (a);         //still prints 42
```

- Now consider the following analogous code using objects (hence references):

```
     //assume we have the following class definition:
     class Point {
          int x, y;
          public Point (int xVal, int yVal) {
               x = xVal;   y = yVal;
          }
     }


...


Point a = new Point (6,4);
Point b = a;
System.out.println (a.x);       // prints 6
b.x = 13;
System.out.println (a.x);       // prints 13 !!
```

TESTING REFERENCES:

- Consider the following code (assume Class Point as before):

```
...
Point p1 = new Point(0,0);
Point p2 = new Point(0,0);  //another point with same values
if (p1 == p2) {
      System.out.println ("The points are equal");
}
```

This will ***not*** print the message; "==" tests if the items (references) are equal

- The following WILL print the message, since the references are equal:

```
...
Point p1 = new Point(0,0);
Point p2 = p1;
if (p1 == p2) {
      System.out.println ("The points are equal");
}
```

- To check if object *contents* are equal, the object must have an "equals()" method:

```
...
if (p1.equals(p2)) {
      System.out.println ("The points are equal");
}
```

- Many Java-defined classes *(e.g. String)* do have "equals()" methods – but not all.

STRING REFERENCES:

- Using and testing **Strings** sometimes causes confusion, but the same rules apply:

```
...
String s1 = new String(“Ed”);
String s2 = new String(“Ed”);    //a different String object
if (s1 == s2) {
      System.out.println (“The strings are equal”);
}
```

  This will ***not*** print the message

- The following examples all WILL print the message:

```
...
if (s1.equals(s2)) {  ...  }

if (s1.equals(“Ed”))  {  ...  }

if (s1.equalsIgnoreCase(“ed”))  {  ...  }
```

  Class String defines both “equals()” and “equalsIgnoreCase()”

- A common mistake:

```
...
if (s1 == “Ed”) {
      System.out.println (“The string contains ‘Ed’ ”);
}
```

  This will ***not*** print the message

- Correct approach:

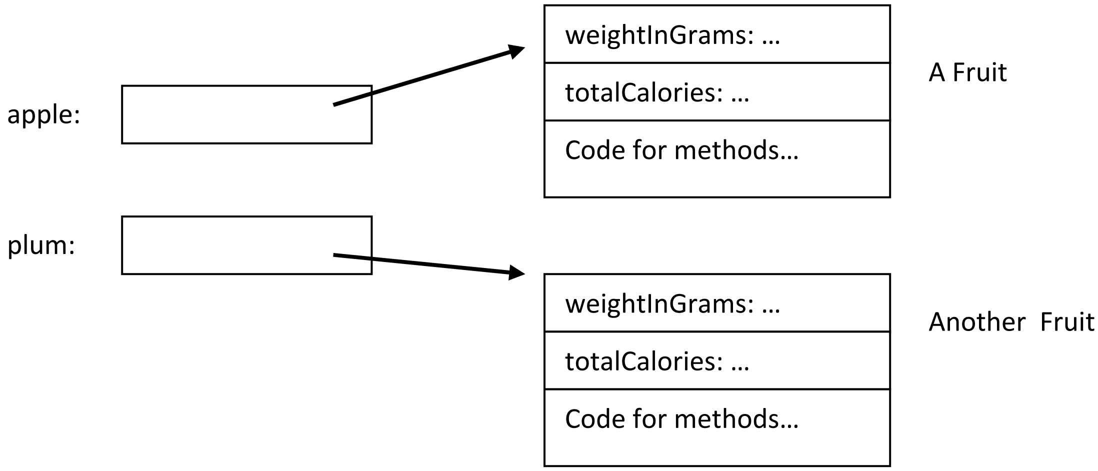
```
if (s1.equals(“Ed”))  { ... }
```

GARBAGE COLLECTION:

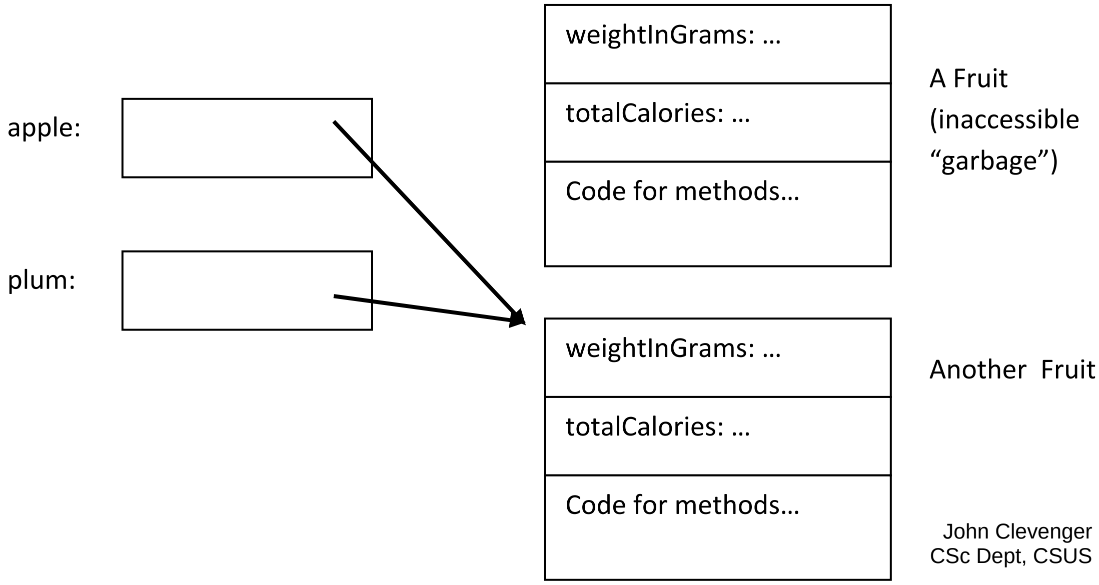
- Important characteristic: assignment does not copy objects; it copies references:

```
Fruit apple = new Fruit();
Fruit plum = new Fruit();
. . .
apple = plum ;        //reference to apple replaced;
                      // now points to same object as “plum”
```

Before assignment:



After assignment:



ARRAYS:

Declaration:

```
int  [ ]  vals ;                 // or:  int  vals  [ ] ;

int  [ ]  scores1, scores2 ;     // two arrays of integers

char  [ ]  grades, letters ;     // two arrays of chars

Point [ ] myPoints ;             // array of objects
```

Characteristics:

- ***Arrays are objects*** (regardless of the type of data in them – primitive or object)
- Like all objects, arrays must be *instantiated:*

```
int  [ ]  vals  = new int [size] ;    //size = # elements

char grades [ ] = new char [5] ;      // five elements

Point [ ] myPoints = new Point [myScreen.getSize()] ;

String [ ] words = new String [25] ;   // holds 25 String refs
```

- Arrays are allocated *on the Heap* (like all objects)
- Size and element-type are fixed at compile time (see “Vector” and “ArrayList” classes)

ARRAYS, cont:

Common Declaration mistakes:

```
int [size] vals ;          //ILLEGAL – need ‘new’

int vals [size];           //ALSO ILLEGAL
```

Indexing:

- Even though they are ***objects***, special syntax allows “normal” indexing:

```
vals [5] = 99 ;                  // store primitives

grades [3] = ‘D’ ;

myPoints [i] = new Point (4,3) ;   // store an object
```

- Indexing range is **0 .. size-1** -- like C
- Runtime range checking is enforced

Arrays as Objects:

- Array names are ***references*** – “pointers to the array object”
- Like all objects, arrays have *FIELDS* and *METHODS*

```
vals.length     // field (not method) giving array size;
                // length is “final” – cannot be changed
```

INITIALIZERS:

```
int [ ] vals  = { 1, 3, 17, 99 } ;

char [ ] letterGrades = { ‘A’, ‘B’, ‘C’, ‘D’, ‘F’ } ;
```

- Implicit instantiation (no *new* needed)
- Size of list determines array size
- Only allowed in a declaration – not runtime assignable:

```
int [ ] vals  = { 1, 3 } ;       // OK

. . .

vals = { 1, 4, 7 } ;       // ILLEGAL (in any form)
```

ARRAYS and REFERENCES:

Potential Confusion:

```
int [ ] myInts  = new int [10] ;

System.out.println (myInts[4]) ;      // prints ‘0’

. . .

myInts[4] = 1776 ;

System.out.println (myInts[4]) ;      // prints ‘1776’
```

but:

```
Ball [ ] myCollection  = new Ball [10] ;

System.out.println (myCollection[4]) ;    // runtime error!

. . .

myCollection [4] = new Ball (Color.blue) ;

System.out.println (myCollection[4]) ;

     // prints string rep of Ball (if rep exists; else error)
```

- Primitives are initialized to data; Object references are initialized to `null`

ARRAYS and REFERENCES, cont:

Another easy “reference” ‘slip-up’:

```
int [ ] a  = { 1, 2, 3 } ;

int [ ] b  = { 1, 2, 3 } ;  //identical data

. . .

if ( a = = b ) {

     System.out.println (“arrays ‘a’ and ‘b’ are equal);

}
```

- The above code does NOT print the message….

- Solution: “**java.util.Arrays**”   [JDK 1.2 (and up)]
  - Contains methods which operate on arrays
  - Method **equals()** does an element-by-element comparison:

```
if ( Arrays.equals(a,b) ) {

     System.out.println ("a and b are equal") ;

}
```

  - Uses “==” for testing primitives
  - Uses (expects) a **.equals()** method to be defined for objects in arrays

ARRAYS OF ARRAYS:

Declaration :

```
int  [ ]  [ ] intTable ;         //”2D array” of ints

Point [ ] [ ] pointTable ;       //”2D array” of Points
```

Instantiation :

```
intTable = new int [3] [5] ;     //could combine with decl.

pointTable = new Point [3] [5] ;
```

Result:

	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0					
Row 1					
Row 2					

Accessing:

```
intTable [0] [2] = 5 ;           //assigns a primitive

pointTable [0] [2] = new Point (17,-6) ; //assigns an object


for (int i=0; i<3; i++) {

     for (int j=0; j<5; j++) {

          intTable [i] [j] =  i * j  ;

          pointTable [i] [j] = new Point (i, j) ;

     }
}
```

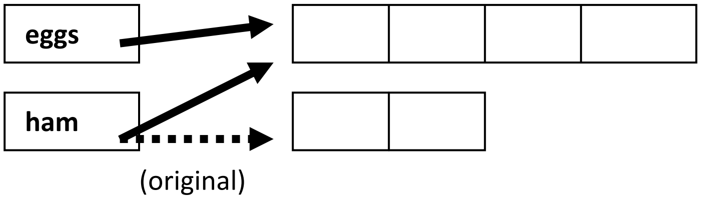
ARRAY ASSIGNMENT:

Arrays can be assigned to another array *if* they have:

- Same element type
- Same number of dimensions:

```
int [ ] eggs  = { 1, 2, 3, 4 } ;

int [ ] ham   = { 1, 2 } ;

. . .

ham = eggs ;    //legal - same element type (int) and dimension

ham [3] = 0 ;   //legal – ham now has 4 elements!
```

- This works because *references* are what is being "assigned" (copied):

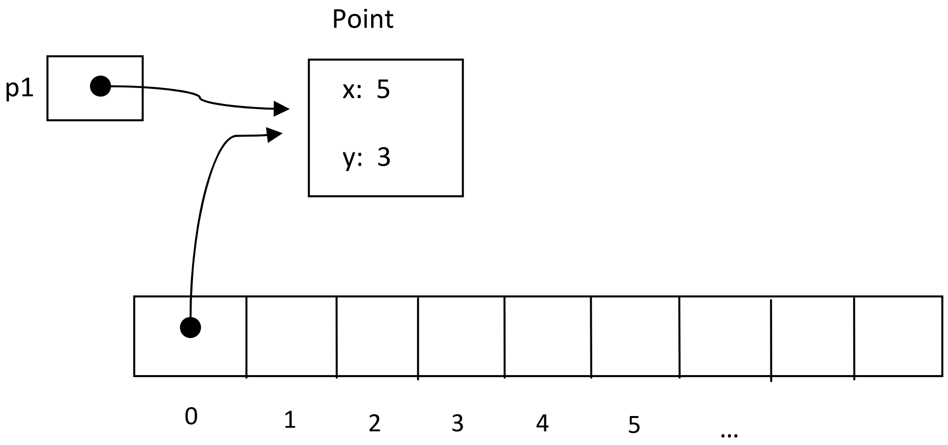


- Method ***System.arraycopy( )*** can be used to perform a real "copy"

Vectors and ArrayLists :

- *Dynamic "arrays"* of *objects*
  - Changeable size
  - Different elements can hold different types
    - Every element is an "Object"
  - **Vectors** are "thread-safe"; **ArrayLists** are not (but are otherwise more efficient)

```
import java.util.Vector ;
. . .
Vector myPoints = new Vector ( );   // create an (empty) vector
. . .
Point p1 = new Point (5,3) ;      // create a Point named p1
myPoints.addElement (p1) ;       // adds a reference to Object p1
```



- A common mistake:

```
. . .
myPoints.addElement (p1) ; // add a point
p1.x ++ ;                  // modify the point
myPoints.addElement (p1) ; // add a new, different point?  NO!
                           // (adds a second reference to
                           //   SAME point (with modified value)
                           //  (i.e. elementAt(0) is changed!!
```

Vectors and ArrayLists (cont.) :

- Advantages of Vectors/ArrayLists:
  - Can grow/shrink dynamically (automatically)
  - Can hold *different object types* in different elements
- Drawbacks of Vectors/ArrayLists:
  - Lose the familiar "indexing" syntax
    - **`myPoints[i]`** becomes **`myPoints.elementAt(i)`**
  - *Slight* space and time penalties over arrays
  - All elements are Objects (instances of Java class "Object")
    - Must type-cast to the appropriate type when retrieving

```
//create some Points and add them to myPoints Vector
. . .
Point p1 = new Point (6,4) ;
myPoints.addElement (p1) ;
. . .

//fetch the 'ith' Point from myPoints Vector
Point nextPoint = myPoints.elementAt (i) ;       // RUNTIME ERROR !!
Point nextPoint = (Point) myPoints.elementAt (i) ;    // correct way
```

Vectors and ArrayLists (cont.) :

- Vectors have *many* methods for manipulating elements (ArrayLists have similar – though not identical – list) :

```
add ( ) ;

addElement ( ) ;

clear ( ) ;

capacity ( ) ;

elementAt ( ) ;

equals ( ) ;

indexOf ( ) ;

insertElementAt ( ) ;

isEmpty ( ) ;

removeElementAt ( ) ;

size ( ) ;

. . .
```

- How do you know what methods exist? And their parameters? And how they work?

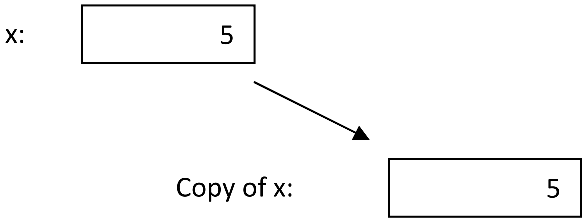
- Answer: **http://java.sun.com/docs/**
  - Complete online Java 2 Standard Edition (J2SE) API documentation
  - Can be downloaded to your machine

PARAMETERS:

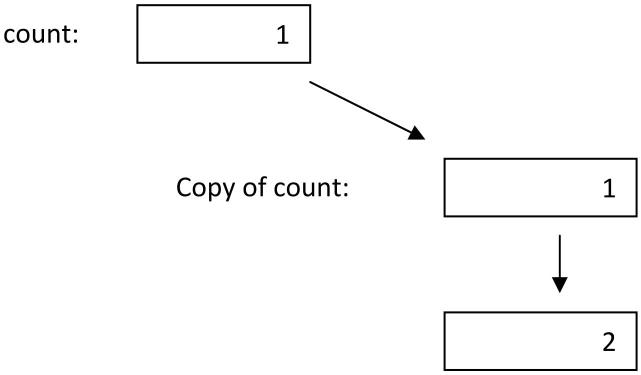
- ALL parameters are passed using “Call by Value” – NEVER “Call by Reference”
    - ➢ A *copy* of the actual value of the parameter is passed
- The original parameter CANNOT be modified by any method to which it is passed
- The *effect* of this *appears* to be different for **primitives** and **objects** (it’s not)

Examples: Primitives

```
int x = 5 ;

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



```
. . .

int count = 1 ;

update (count) ;

System.out.println (count) ;

. . .

=============================


public void update (int count) {

      . . .

      count = count + 1 ;

      System.out.println (count) ;
}
```



PARAMETERS, cont:

- Parameters which are *objects* are also passed "by value" -- i.e. a *copy* is passed
- The original parameter still CANNOT be modified by any method to which it is passed
- But: objects are represented by *references:* a copy of the reference is passed
  - Result: the receiving method cannot alter the original *reference – but it can alter the object itself* (since it has a reference to it)

Examples: Objects:

```
Ball myBall = new Ball (Color.red);
. . .
System.out.println (myBall.color) ;   //presumes "color" is
                                      // public in Ball
display (myBall) ;
     System.out.println (myBall.color) ;
. . .

     =================================


public void display (Ball theBall) {

     System.out.println (theBall.color) ;

     theBall.color = Color.blue ;

}
```

Java vs. C++[†]

Java has:

- **No "preprocessor" (hence no `#include`, .h files, `#define`, etc…)**
- **No "global variables"**
- **Fixed (defined) sizes for primitives (e.g., `int` is *always* 32 bits)**
- **No Pointers. "References" are similar, but:**
  - **Cannot be converted to a primitive**
  - **Cannot be manipulated with arithmetic operators**
  - **Have no "&" (address-of) or dereference (" * " or " -> ") operators**
- **Automatic garbage collection**
  - **Objects which cannot be accessed (are "out of scope" and have no copied references) are automatically returned to the "heap"**
- **No "`goto`" statement**
- **No "`struct`" or "`union`" types**
- **No "function pointers" (although this can be simulated by passing objects which implement a given interface)**
- **No support for multiple inheritance of method implementation**
- **A weaker form of `templates` (called `generics`) based on a notion called `type erasure`**
- **No support for operator overloading**

[†] Excerpted from Java In A Nutshell, David Flanagan, O'Reilly

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

III - Object Oriented Concepts

- Overview – the OOP “PIE”
- Abstraction
- Encapsulation
  - Bundling
  - Information Hiding
  - Implementing Encapsulation
  - Accessors
- UML Class Diagrams
- Associations
  - Aggregation
  - Composition
  - Dependencies
  - Implementing Associations

CSc 133 Lecture Notes

III - OOP Concepts

Computer Science Dept.

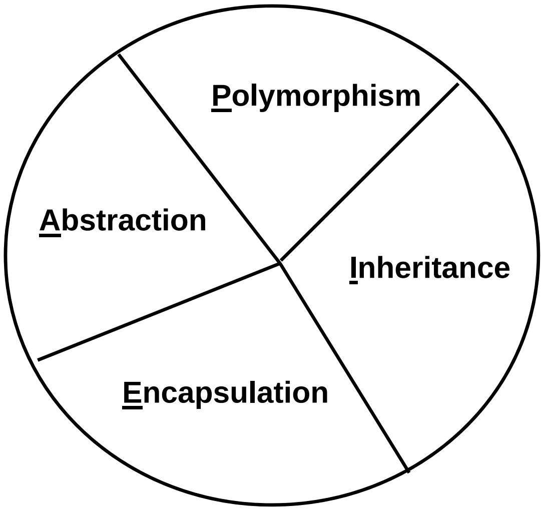
CSUS

Overview

- Abstraction
- Encapsulation
  - Accessors & Visibility
- UML Class Diagrams
- Class Associations
  - Aggregation
  - Composition
  - Dependency

The OOP “Pie”

- Four distinct OOP Concepts make “A PIE”



Polymorphism

Abstraction

Inheritance

Encapsulation

Abstraction

- Identification of the minimum essential characteristics of an entity
- Essential for specifying (and simplifying) large, complex systems
- OOP supports:
  - *Procedural* abstraction (objects have functions)
  - *Data* abstraction (attributes can be objects)
- “Abstract Data Types” (example: “Stack”)

Encapsulation

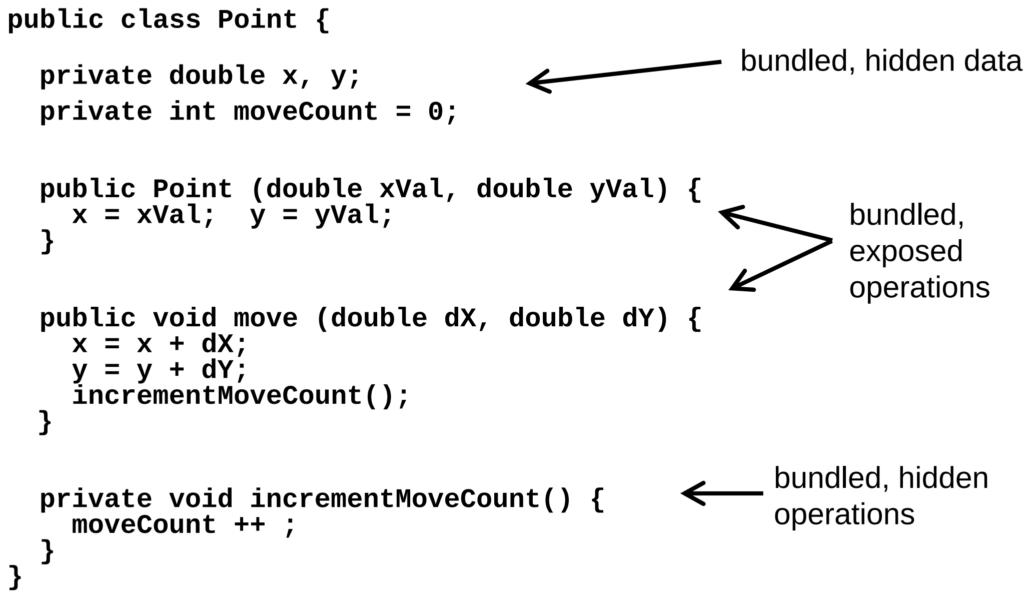
"Bundling"

- Collecting together the data and operations associated with an abstraction
- CLASS: attributes (fields) and functions (methods)

"Information Hiding"

- Separating details of implementation from user access
- Public vs. Protected vs. Private
- Accessors: *Selectors* vs. *Mutators*

Implementing Encapsulation



```
public class Point {

  private double x, y;
  private int moveCount = 0;

  public Point (double xVal, double yVal) {
    x = xVal;  y = yVal;
  }

  public void move (double dX, double dY) {
    x = x + dX;
    y = y + dY;
    incrementMoveCount();
  }

  private void incrementMoveCount() {
    moveCount ++ ;
  }
}
```

bundled, hidden data

bundled, exposed operations

bundled, hidden operations

Abstraction example: Color

- Three axes: Red, Green, Blue
- Distance along axis = intensity (0 to 255)
- Locations within cube = different colors
  - Values of equal RGB intensity are grey

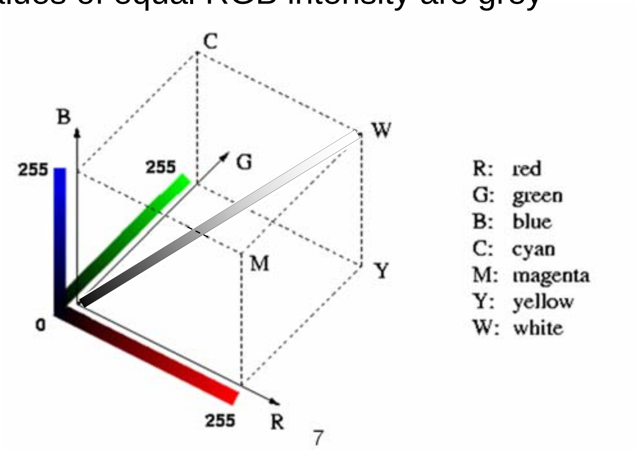


Image credit: http://gimp-savvy.com

Example: Java Color Class

- An *encapsulated abstraction*
- Uses “RGB color model”
- Multiple constructors:
  - `Color myRed = new Color (1.0, 0.0, 0.0);`
  - `Color myGreen = new Color (0, 255, 0);`
- Has numerous *accessors* (no *mutators*):

  getRed() getGreen() getBlue()

- Has static *constants* for many colors
  - `Color myRed = Color.red;`
  - `Color myGreen = Color.GREEN;`

Breaking Encapsulations

- **The wrong way, with public data:**

```
public class Point {
  public double x, y;

  public Point () {
    x = 0.0 ;   y = 0.0 ;
  }

  // other methods here...
 }
```

BAD!

Breaking Encapsulations (cont.)

- **The correct way, with “Accessors”:**

```
public class Point {

   private double x, y ;

   public Point () {
     x = 0.0 ;    y = 0.0 ;
   }

   public double getX() {
     return x ;
   }

   public double getY() {
     return y ;
   }

   public void setX (double newX) {
     x = newX ;
   }

   public void setY (double newY) {
     y = newY ;
   }

   // etc.
}
```

Note

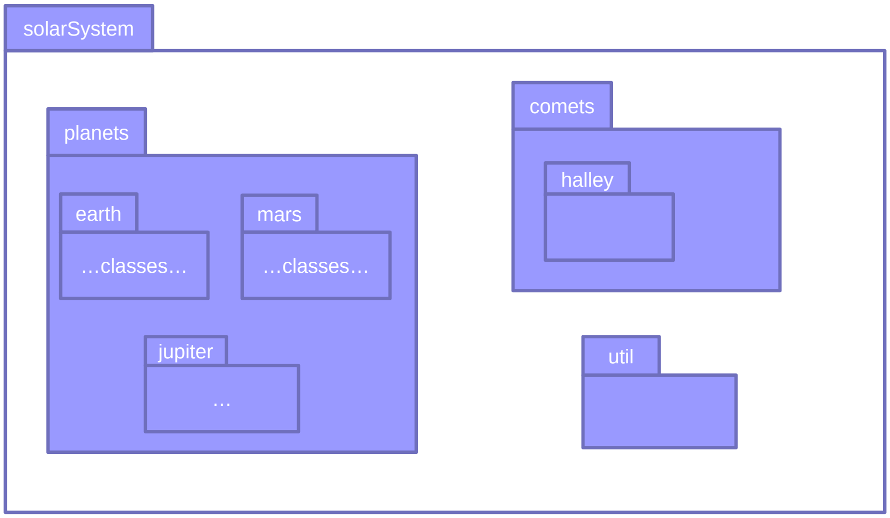
Access (Visibility) Modifiers

	Modifier	Access Allowed By			
		Class	**Package**	**Subclass**	**World**
Java:	public	Y	Y	Y	Y
	protected	Y	Y	Y	N
	<none>	Y	Y*	N	N
	private	Y	N	N	N
C++:	public	Y	<n/a>	Y	Y
	protected	Y	<n/a>	Y	N
	<none>	Y	<n/a>*	N	N
	private	Y	<n/a>	N	N

*In C++, omitting any visibility specifier is the same as declaring it *private*, whereas in Java this allows *"package access"*

Java Packages

- Used to group together classes belonging to the same category or providing similar functionality



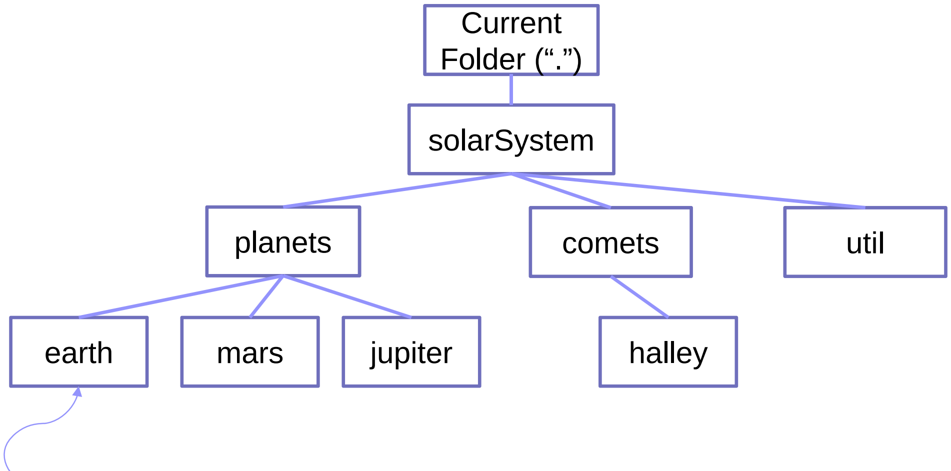
Java Packages (cont.)

- Packages are *named* using the concatenation of the enclosing package names
- Types (e.g. classes) must declare what package they belong to
  - Otherwise they are placed in the "default" (unnamed) package
- Package names become part of the class name; the following class has the full name *solarSystem.planets.earth.Human*

```
package solarSystem.planets.earth ;

//a class defining species originating on Earth
public class Human {

  // class declarations and methods here...
}
```

Packages and Folders

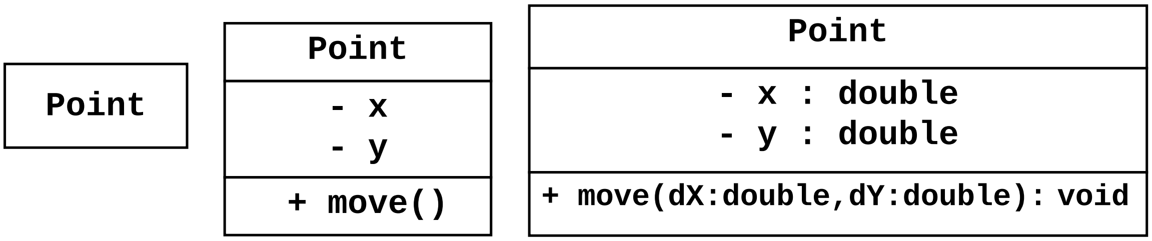
- Classes reside in (are compiled into) *folder hierarchies* which match the package name structure:



Class "Human" (inside folder "earth")

UML “Class Diagrams”

- Unified Modeling Language defines a “graphical notation” for classes
  - UML for the “**Point**” class:



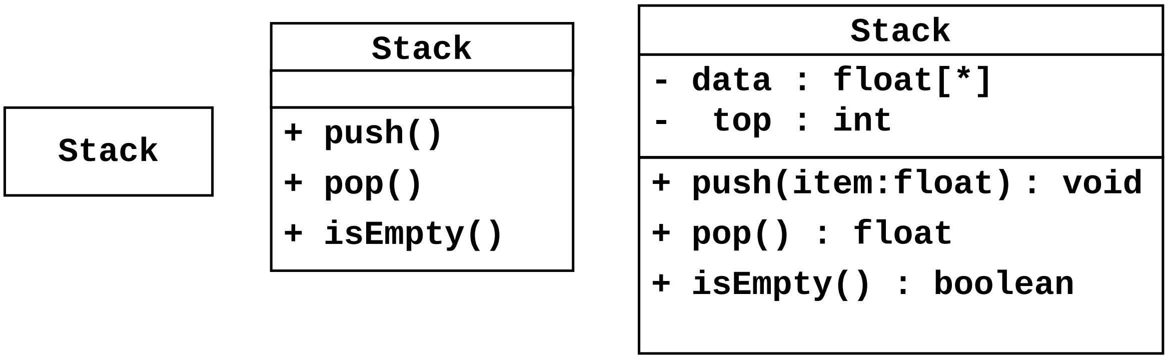
Point

Point
- x
- y
+ move()

Point
- x : double
- y : double
+ move(dX:double,dY:double): void

UML “Class Diagrams” (cont.)

- UML for the “**Stack**” class:



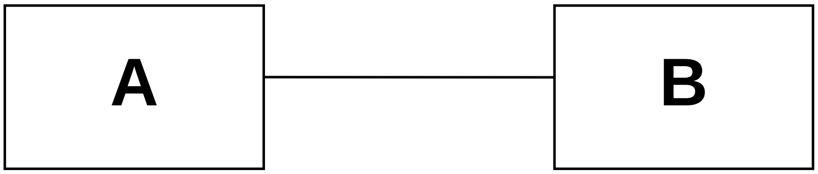
Stack

Stack
+ push()
+ pop()
+ isEmpty()

Stack
- data : float[*]
- top : int
+ push(item:float) : void
+ pop() : float
+ isEmpty() : boolean

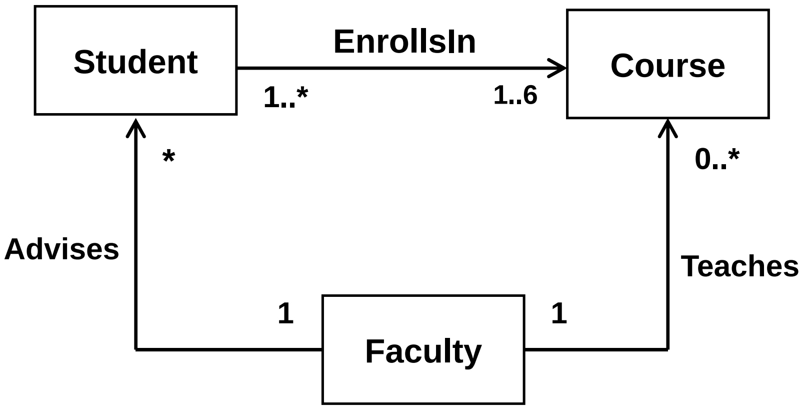
Associations

- Definition: An *association* exists between two classes A and B if instances can send or receive messages (make method calls) between each other.



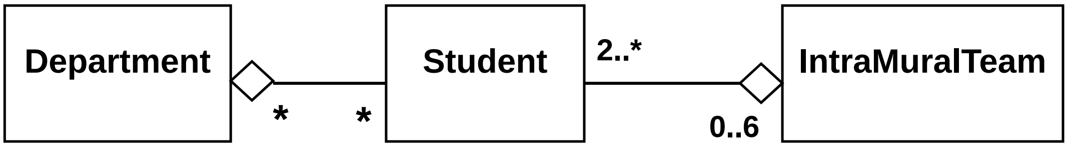
Associations (cont.)

- Associations can have properties:
    - Cardinality
    - Direction
    - Label (name)



Special Kinds Of Associations

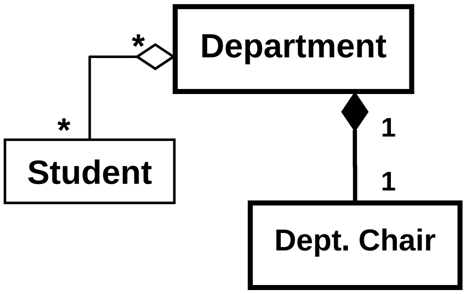
- **Aggregation**
  - Represents ***"has-a"*** or ***"is-Part-Of"***



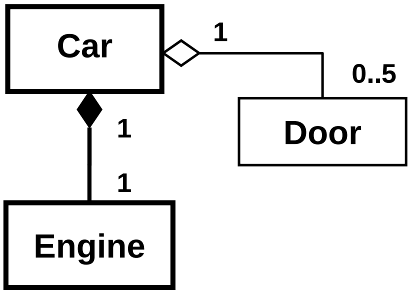
- **An IntraMuralTeam is an aggregate of *(has)* 2 or more Students**
- **A Student *is-a-part-of* at most six Teams**
- **A Department has any number of Students**
- **A Student can belong to any number of Departments (e.g. double major)**

Special Kinds Of Associations (cont.)

- Composition : a *special type* of *aggregation*
- Two forms:
  - "exclusive ownership" (without whole, the part can't exist)
  - "required ownership" (without part, the whole can't exist)



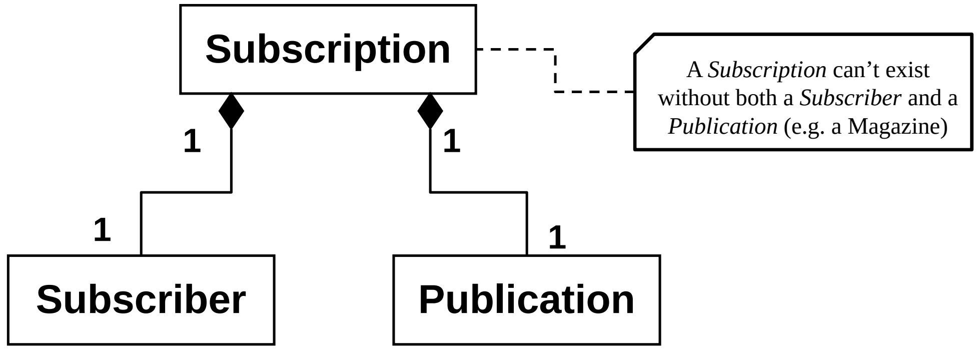
Exclusive ownership



Required ownership

Special Kinds Of Associations (cont.)

- Composition (another example)

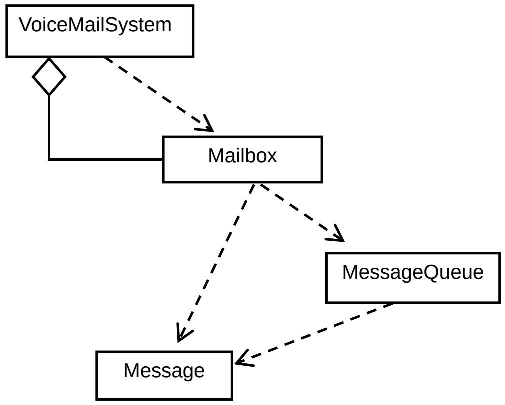


A *Subscription* can't exist without both a *Subscriber* and a *Publication* (e.g. a Magazine)

Special Kinds Of Associations (cont.)

- Dependency
  - **Represents "<u>uses</u>" (or "knows about")**

- **Indicates *coupling* between classes**
- **Desireable to *minimize* dependencies**
- **Other relationships (e.g. aggregation, inheritance) *imply dependency***

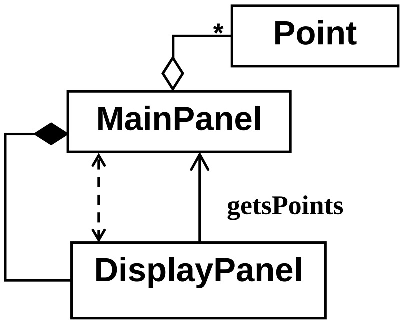


Implementing Associations

- Associations can be unary or binary
- Links are stored in private attributes

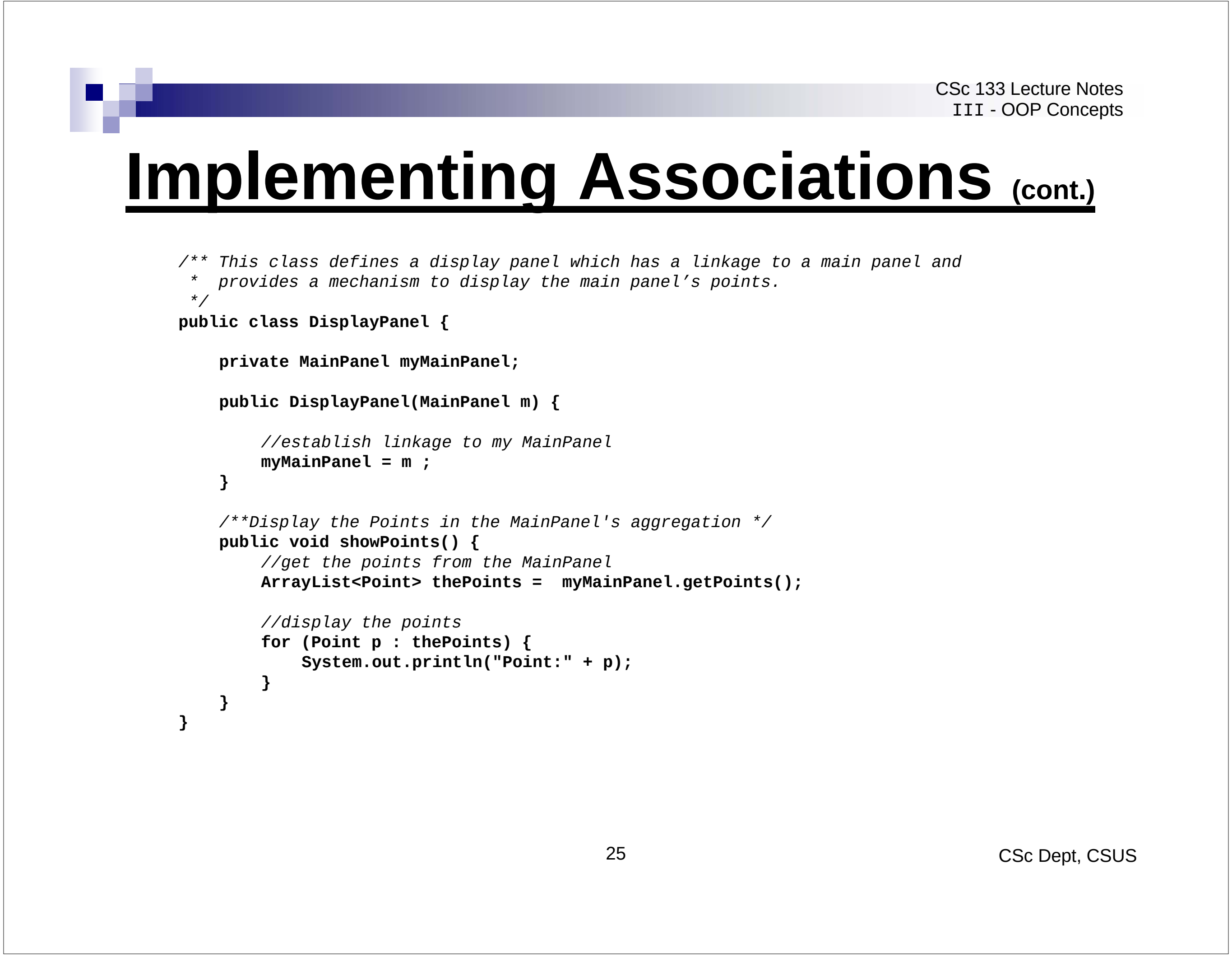
```
public class MainPanel {
  private DisplayPanel myDisPanel = new DisplayPanel (this) ;
  ...
}
```

```
public class DisplayPanel {
   private MainPanel myMainPanel ;

   //constructor receives and saves reference
   DisplayPanel (MainPanel theMainPanel) {
     myMainPanel = theMainPanel ;
   }
   ...
}
```



Implementing Associations (cont.)

```
/**This class defines a "MainPanel" with the following Class Associations:
 *  -- an aggregation of Points  -- a composition of a DisplayPanel.
 */
public class MainPanel {

    private ArrayList<Point> myPoints ;     //my Point aggregation
    private DisplayPanel myDisplayPanel;    //my DisplayPanel composition

    /** Construct a MainPanel containing a DisplayPanel and an
     *  (initially empty) aggregation of Points. */
    public MainPanel () {
        myDisplayPanel = new DisplayPanel(this);
    }

    /**Sets my aggregation of Points to the specified collection */
    public void setPoints(ArrayList<Point> p) { myPoints = p; }

    /** Return my aggregation of Points */
    public ArrayList<Point> getPoints() { return myPoints ; }

    /**Add a point to my aggregation of Points*/
    public void addPoint(Point p) {
        //first insure the aggregation is defined
        if (myPoints == null) {
            myPoints = new ArrayList<Point>();
        }
        myPoints.add(p);
    }
}
```



Implementing Associations (cont.)

```
/** This class defines a display panel which has a linkage to a main panel and
 *  provides a mechanism to display the main panel’s points.
 */
public class DisplayPanel {

    private MainPanel myMainPanel;

    public DisplayPanel(MainPanel m) {

        //establish linkage to my MainPanel
        myMainPanel = m ;
    }

    /**Display the Points in the MainPanel's aggregation */
    public void showPoints() {
        //get the points from the MainPanel
        ArrayList<Point> thePoints =  myMainPanel.getPoints();

        //display the points
        for (Point p : thePoints) {
            System.out.println("Point:" + p);
        }
    }
}
```

<< This page intentionally left (almost) blank >>

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

IV – Inheritance

- **Definition**
- **Representation in UML**
- **Implementation in Java**
- **The “IS-A” concept**
- **Inheritance Hierarchies**
- **Overriding**
- **Overloading**
- **Implications for Public vs. Private data**
- **Forms of Inheritance**
- **Abstract classes and methods**
- **Single vs. Multiple Inheritance**

CSc 133 Lecture Notes

IV - Inheritance

Computer Science Dept.
CSUS

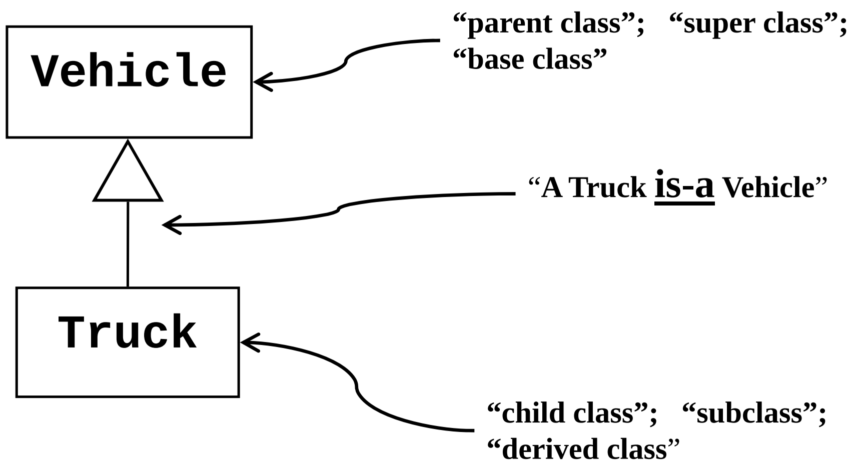
Overview

- Definition
- Inheritance hierarchies
- Overriding
- Forms of Inheritance:
  - Extension
  - Specialization
  - Specification
- Single vs. Multiple inheritance
- Implementation details

What Is Inheritance?

- **A specific kind of *association* between classes**
- **Various definitions:**
  - Creation of a *hierarchy of classes*, where lower-level classes share properties of a common "parent class"
  - A mechanism for indicating that one class is "similar" to another but has specific differences
  - A mechanism for enabling properties (attributes and methods) of a "super class" to be propagated down to "sub classes"
  - Using a "base class" to define what characteristics are *common* to all instances of the class, then defining "derived classes" to define what is special about each subgrouping

Inheritance In UML



"parent class"; "super class"; "base class"

"A Truck is-a Vehicle"

"child class"; "subclass"; "derived class"

Inheritance In Java

- **Specified with the keyword "<u>extends</u>" :**

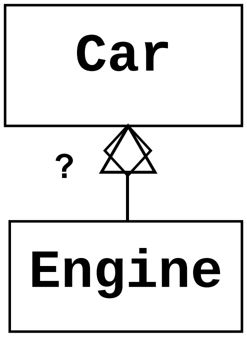
```
public class Vehicle {

  private int weight;
  private double purchasePrice;
  //... other Vehicle data here

  public Vehicle ()
  { ... }

  public void turn (int direction)
  { ... }

  // ... other Vehicle methods here
}
```

```
public class Truck extends Vehicle {
  private int freightCapacity;
  //... other Truck data here

  public Truck ()
  { ... }

  // ... Truck-specific methods here
}
```

- **Note: a Truck "is-a" Vehicle**
- **Only a <u>single</u> "extends" allowed (no "multiple inheritance")**
- **Absence of any "extends" clause implies "<u>extends Object</u>"**

The "IS-A" Relationship

- **Inheritance *<u>always</u>* specifies an "<u>is-a</u>" relationship.**
- **If you can't say "A is a B" (or "A is a kind of B"), *<u>it isn't inheritance</u>***



An Engine "is a" Car ? X

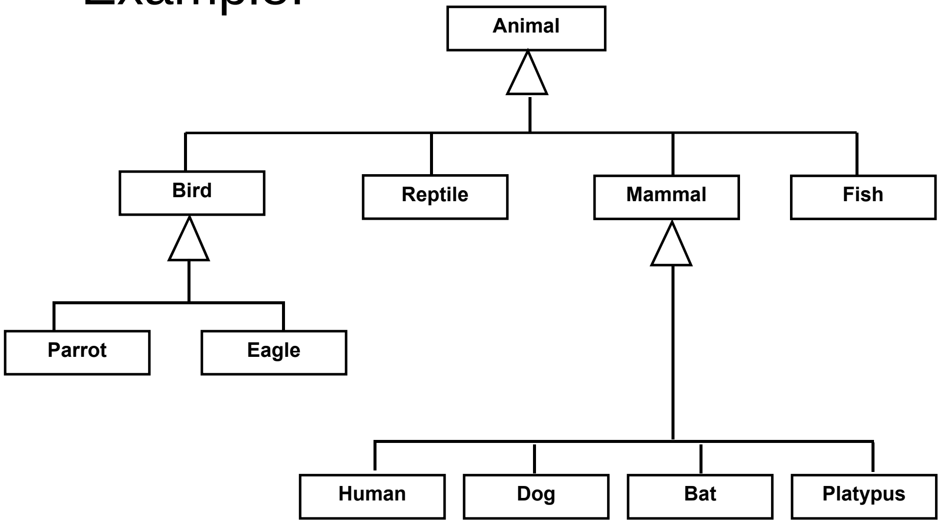
A Car "is an" Engine ? X

A Car "<u>has-an"</u> Engine ✓

An Engine "<u>is a part of</u>" a Car ✓

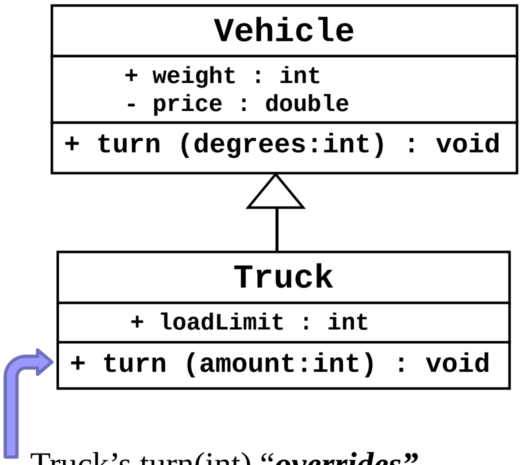
Inheritance Hierarchies

- Example:



Method Overriding

- Inheritance leads to an interesting possibility:
  duplicate method declarations



```
public class Vehicle {
  public int weight ;
  private double price ;

  public void turn (int degrees)
  { // some code to accomplish turning... }
  ...
}
```

```
public class Truck extends Vehicle {
  public int loadLimit ;

  public void turn (int amount)
  { // different code to accomplish turning... }
  ...
}
```

Truck's turn(int) "***overrides***"

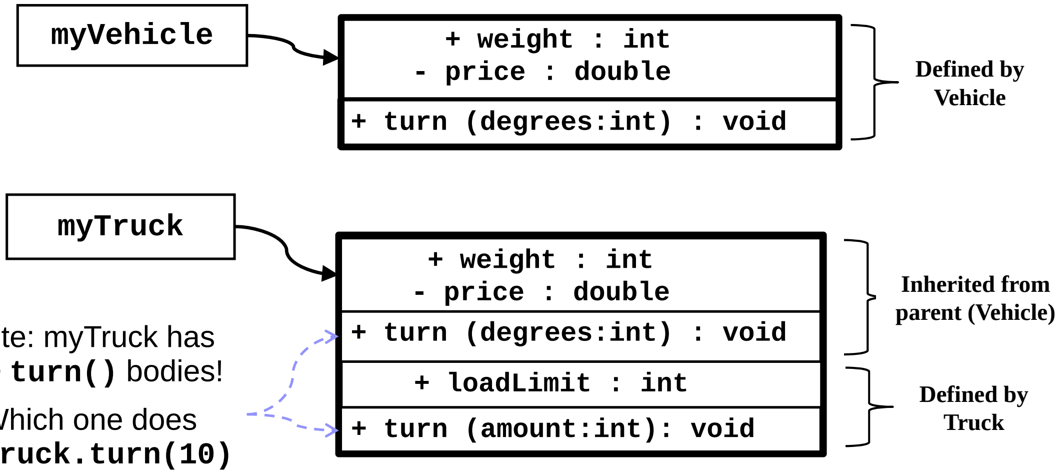
Vehicle's turn(int)

Effects of Method Overriding

Consider the following code:

```
Vehicle myVehicle = new Vehicle();
Truck myTruck = new Truck();
```

… then we get two objects:



Note: myTruck has *two* **`turn()`** bodies!

Which one does **`myTruck.turn(10)`** invoke?

Method Overriding: Summary

- **Occurs when a child class redefines an inherited method, with:**
  - **same name**
  - **same parameters *("signature")***
  - **same return type**
- **Child objects contain the code for both methods**
  - **Parent method code plus the child (overriding) method code**
- **Calling an overridden method (in Java) invokes the child version**
  - **Never invokes the parent version**
  - **The child can invoke the parent method using "super.*xxx (...)*"**
- **It is not legal (in Java) to override and change the *return type*.**
  - **So for the Vehicle/Truck example, Truck could NOT define**

```
public boolean turn (int amount) { ... }
```

Overloading

- ***Not*** **the same as “overriding”…**
  - **Overloading == same name but different signatures**
  - **Can occur *in the same class or split between parent/child classes***

- ***Overloading examples:***
  - Methods with different numbers of parameters:

    ```
    distance(p1);    distance(p1,p2)
    ```

  - Constructors with different parameter sequences:

    ```
    Ball();   Ball(Color c);   Ball (int radius);
    Ball(Color c, int radius);
    ```

  - Changing parameter type:

    ```
    computeStandings(int numTeams);
    computeStandings(double average);
    computeStandings(Hashtable teams);
    ```

recall, from the encapsulation section:

Point (without “Accessors”):

```
public class Point {
   public double x, y ;
   public Point () {
     x = 0.0 ;
     y = 0.0 ;
   }
}
```

BAD

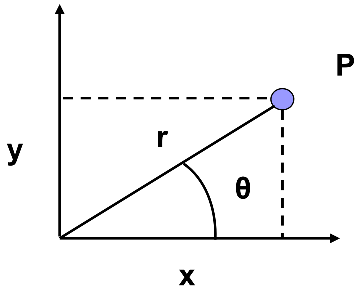
Now we will learn why!

Point (with “Accessors”):

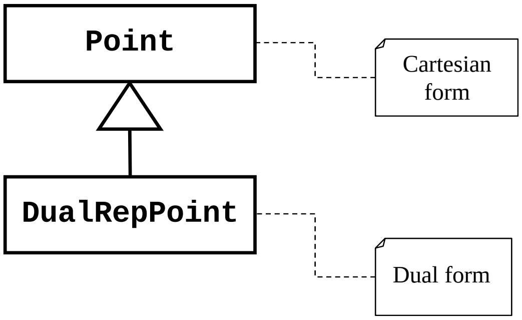
```
public class Point {
   private double x, y ;
   public Point (){
      x = 0.0 ;
      y = 0.0 ;
   }
   public double getX() {
      return x ;
   }
   public double getY() {
      return y ;
   }
   public void setX (double newX) {
      x = newX ;
   }
   public void setY (double newY) {
      y = newY ;
   }
}
```

GOOD

Example: extend “**Point**” to create “**DualRepPoint**”



P(x,y) = P(r,θ)



Point

DualRepPoint

Cartesian form

Dual form

DualRepPoint (DRP): Ver. 1

```
public class DualRepPoint extends Point {

  public double radius, angle ;                     ← Note public access

  /** Constructor: creates a default point with radius 1 at 45 degrees */
  public DualRepPoint () {
    radius = 1.0 ;
    angle = 45 ;
    updateRectangularValues();
  }

  /** Constructor: creates a point as specified by the input parameters */
  public DualRepPoint (double theRadius, double angleInDegrees) {
    radius = theRadius ;
    angle = angleInDegrees;
    updateRectangularValues();
  }

  /** Force the Cartesian values (inherited from Point) to be consistent */
  private void updateRectangularValues() {
    x = radius * Math.cos(Math.toRadians(angle));    // legal assignments
    y = radius * Math.sin(Math.toRadians(angle));    // (x & y are public)
  }
}
```

Client Using Public Access

```
/** This shows a "client" class that makes use of the "V. 1 DualRepPoint" class.
 *  It shows how the improper implementation of DualRepPoint (that is, use of
 *  fields with public access) leads to problems...
 */
public class SomeClientClass {

  private DualRepPoint myDRPoint ;  //declare client's local DualRepPoint

  // Constructor: creates a DualRepPoint with default values,
  // then changes the DRP's radius and angle values

  public SomeClientClass() {
      myDRPoint = new DualRepPoint() ;     //create private DualRepPoint
      myDRPoint.radius = 5.0 ;             //update myPoint's values
      myDRPoint.angle = 90.0 ;
  }
  ...
}
```

Anything wrong?

DualRepPoint: Ver. 2

```
/** This class maintains a point representation in both Polar and Rectangular
 *  form and protects against inconsistent changes in the local fields */
public class DualRepPoint extends Point {

  private double radius, angle ;

  // constructors as before (not shown) ...
  public double getRadius() { return radius ; }
  public double getAngle() { return angle ; }

  public void setRadius(double theRadius) {
    radius = theRadius ;
    updateRectangularValues();
  }

  public void setAngle(double angleInDegrees) {
    angle = angleInDegrees;
    updateRectangularValues();
  }

  // force the Cartesian values (inherited from Point) to be consistent
  private void updateRectangularValues() {
    x = radius * Math.cos(Math.toRadians(angle));
    y = radius * Math.sin(Math.toRadians(angle));
  }
}
```

← **New: private access**

New: public accessors

Client Using DRP Accessors

```
/** This new version of the client code shows how requiring the use of accessors
 *  when manipulating the DualRepPoint radius & angle fields fixes (one) problem ...
 */

public class SomeClientClass {

  private DualRepPoint myDRPoint ;

  public SomeClientClass() {                 // client constructor
      myDRPoint = new DualRepPoint();        // create a private DualRepPoint
      myDRPoint.setRadius(5.0) ;  // alter DRP's values (safely): client has
      myDRPoint.setAngle(90.0) ;  // no way to access radius/angle directly
  }

  .... etc.
}
```

Problem solved?

Accessing Other DRP Fields

```
/** This newer version of the client code shows how requiring the use of accessors
 *  when manipulating the DualRepPoint radius & angle fields fixes (one) problem
 *  ... but not all problems...
 */
public class SomeClientClass {

  private DualRepPoint myDRPoint ;

  public SomeClientClass() {                 // client constructor as before
      myDRPoint = new DualRepPoint();
      myDRPoint.setRadius(5.0) ;
      myDRPoint.setAngle(90.0) ;
  }

  //a new client method which manipulates the portion inherited from Point
  public void someMethod() {
      myDRPoint.x = 2.2 ;
      myDRPoint.y = 7.7 ;
      ...
  }
  ... etc.
}
```

Anything wrong?

Public Fields *Break Code*

- Point (without “Accessors”):

```
public class Point {
      public double x, y ;
      public Point () {
        x = 0.0 ;
        y = 0.0 ;
      }
         . . .
   }
```

BAD BAD BAD

Using Accessors

- Point (with “Accessors”):

```
public class Point {
   private double x, y ;
   public Point (){
     x = 0.0 ;
     y = 0.0 ;
   }
   public double getX() { return x ; }
   public double getY() { return y ; }
   public void setX (double newX) {
     x = newX ;
   }
   public void setY (double newY) {
     y = newY ;
   }
   // other methods here...
}
```

Good ! Good ! Good ! Good !

Accessors Don't Solve All Problems

```
/** This new version of the client code shows how requiring the use of accessors
 *  in ALL classes may have fixed ONE problem ... but another still exists
 */

public class SomeClientClass {
  private DualRepPoint myDRPoint ;
  public SomeClientClass() {               // client constructor
      myDRPoint = new DualRepPoint();      // create a private DualRepPoint
      myDRPoint.setX(2.2) ;       // alter DRP's inherited X,Y values
      myDRPoint.setY(7.7) ;       //   using inherited accessors
  }
  .... etc.
}
```

- **Problem still exists!**
- **Solution ?**

DualRepPoint: Correct Version

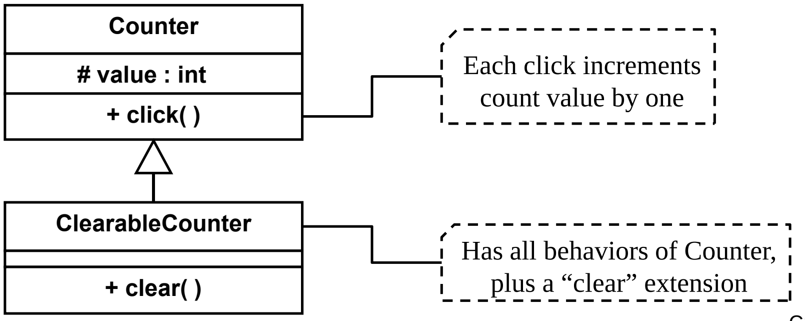
```
public class DualRepPoint extends Point  {   //uses "Good" Point with accessors
  private double radius, angle ;
  //...constructors and accessors for radius and angle here as before ...
  // Override inherited accessors
  public void setX (double xVal) {  //note that overriding the parent accessors
    super.setX(xVal) ;              // makes it impossible for a client to put
    updatePolarValues() ;           // put a DRP into an inconsistent state
  }
  public void setY (double yVal) {
    super.setY(yVal) ;
    updatePolarValues() ;
  }
  private void updateRectangularValues() {
    super.setX(radius * Math.cos(Math.toRadians(angle))) ;
    super.setY(radius * Math.sin(Math.toRadians(angle))) ;
  }
  //new private method to maintain consistent state
  private void updatePolarValues() {
    double x = super.getX() ;     //note: some people would use protected to
    double y = super.getY() ;     // to allow direct subclass access to X & y
    radius = Math.sqrt (x*x + y*y) ;
    angle = Math.atan2 (y,x) ;
  }
}
```

Typical Uses for Inheritance

- **Extension**
  - **Define *new behavior,* and**
  - **Retaining existing behaviors**
- **Specialization**
  - **Modify existing behavior(s)**
- **Specification**
  - **Provide ("specify") the implementation details of "abstract" behavior(s)**

Inheritance for Extension

- **Used to *define new behavior***
  - **Retains parent class' Interface and implementation**
- **Example: Counter**
  - **Base class increments on each "click"**
  - **Extension adds support for "clearing" (resetting)**



Each click increments count value by one

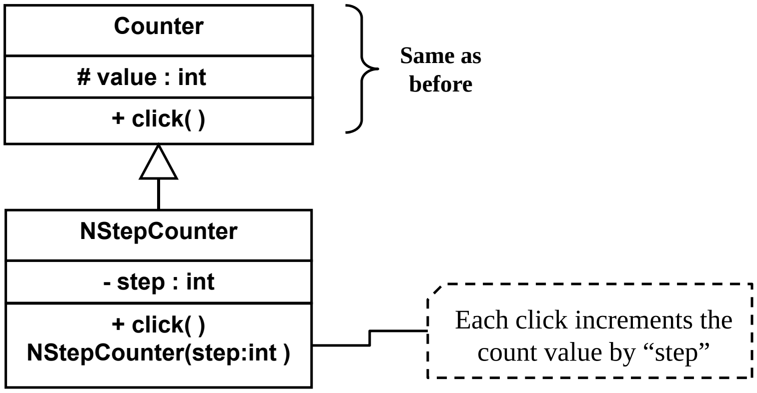
Has all behaviors of Counter, plus a "clear" extension

Inheritance for Extension (cont.)

```
/** This class defines a counter which increments on each call to click().
 * The Counter has no ability to be reset.  */
public class Counter {
  protected int value ;

  /** Increment the counter by one. */
  public void click() {
    value = value + 1;
  }
}

/** This class defines an object with all the properties of a Counter, and
 *  which also has a "clear" function to reset the counter to zero. */
public class ClearableCounter extends Counter {

  // Reset the counter value to zero.  Note that this method can
  // access the "value" field in the parent because that field
  //  is defined as "protected".

  public void clear () {
    value = 0 ;
  }
}
```

Inheritance for Specialization

- Used to *modify* ***existing*** *behavior* (i.e. behavior defined by parent)
- Retains parent class' ***interface***
- Uses ***overriding*** to change the behavior
- Example: N-Step Counter

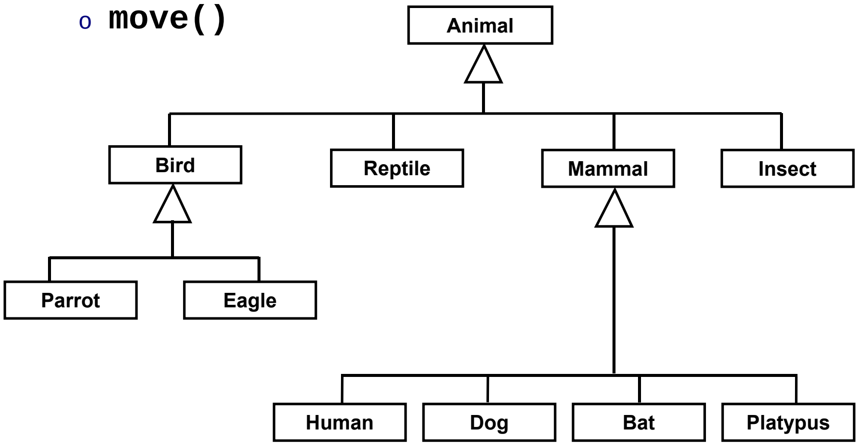


Inheritance for Specification

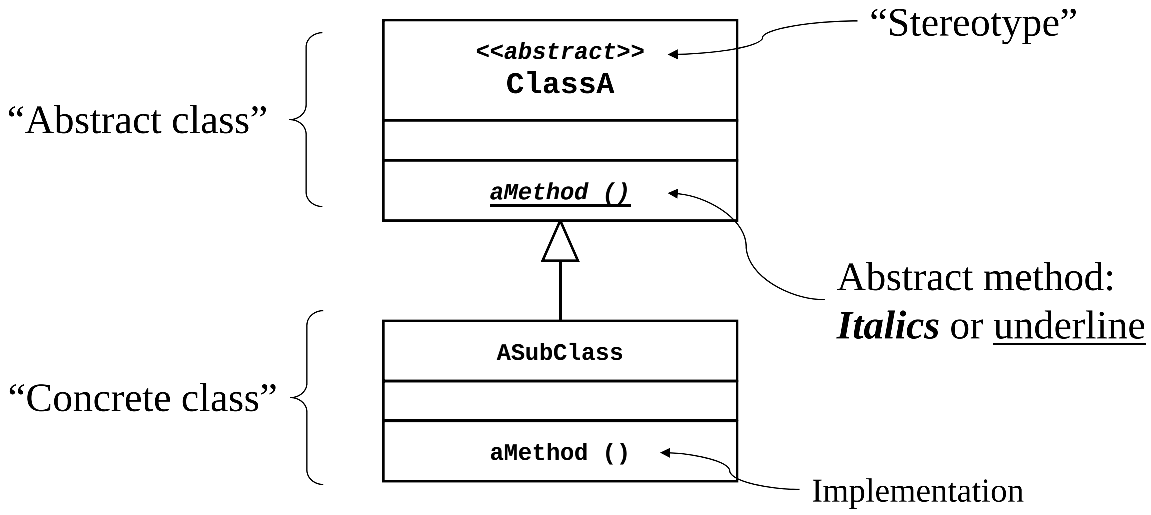
- Used to *specify (define)* behavior ***declared*** (but not ***defined***) by the parent
  - Classes which declare but don't define behavior: **Abstract Classes**
  - Methods which don't contain implementations: ***Abstract methods***

Abstract Classes & Methods

- Some classes will never logically be instantiated
  - **`Animal`**, **`Mammal`**, …
- Some methods cannot be "specified" completely at a given class level
  - **`move()`**



Inheritance for Specification (cont.)


“Stereotype”

“Abstract class”

<<abstract>>
ClassA

aMethod ()

Abstract method:
Italics or <u>underline</u>

“Concrete class”

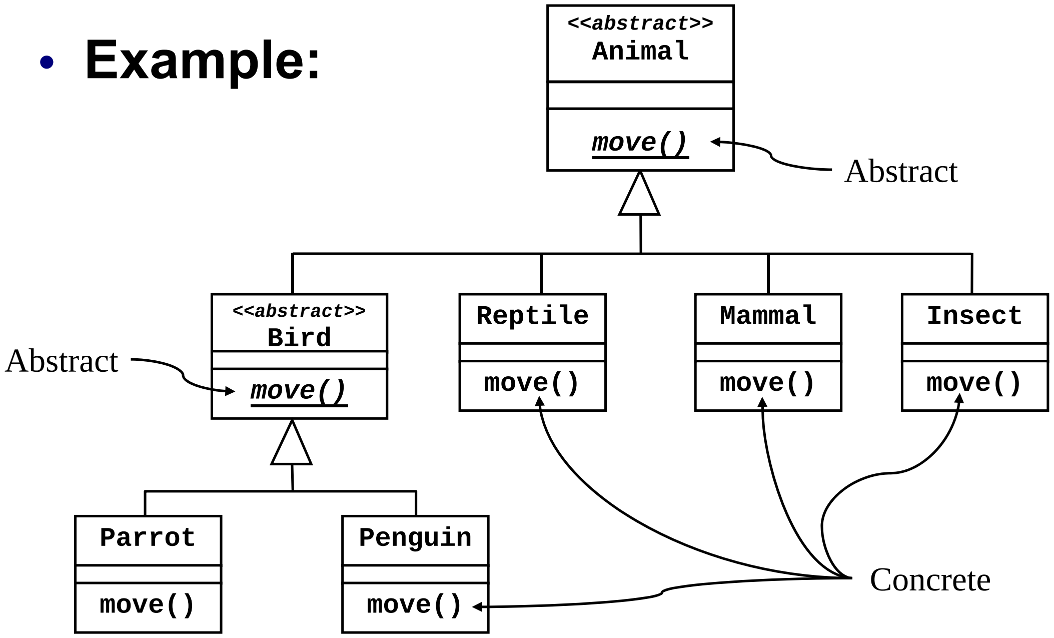
ASubClass

aMethod ()

Implementation

Inheritance for Specification (cont.)

- **Example:**


<<abstract>>
Animal

move()

Abstract

<<abstract>>
Bird

move()

Abstract

Reptile

move()

Mammal

move()

Insect

move()

Parrot

move()

Penguin

move()

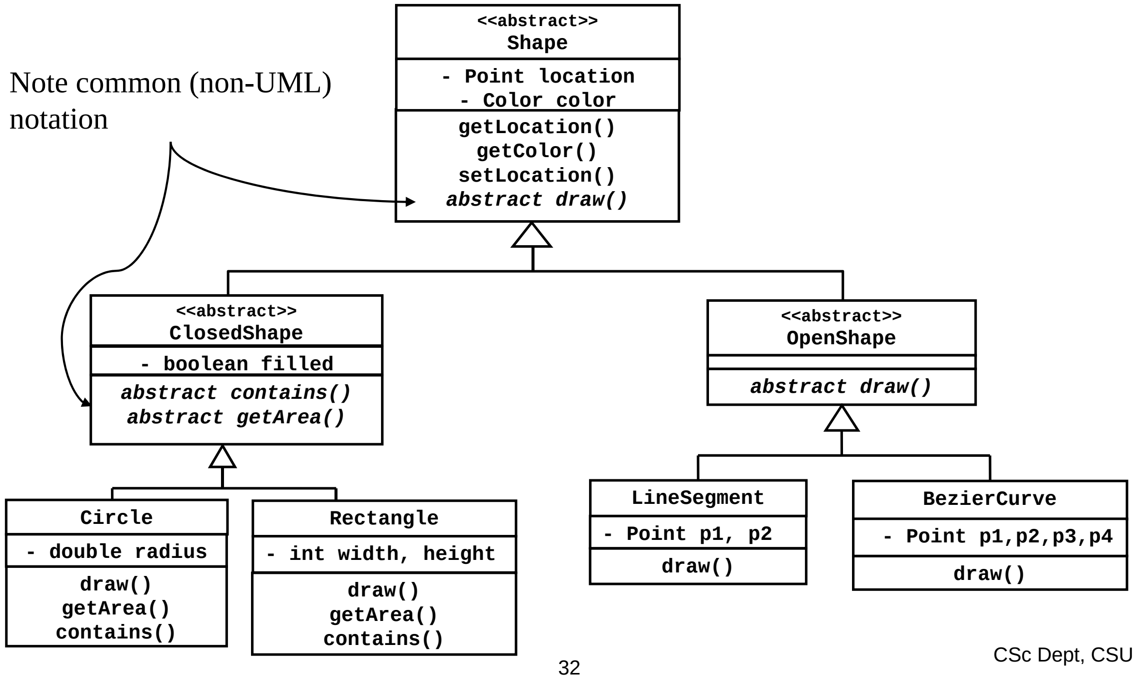
Concrete

Inheritance for Specification (cont.)

- **Another example: abstract shapes**
  - Different kinds of shapes:
    - **`Line   Circle   Rectangle   BezierCurve   ...`**
  - Common (shared) characteristics :
    - a “Location”
    - a Color
    - …
  - Common operations (methods) :
    - **`getLocation()`**
    - **`setLocation()`**
    - **`getColor()`**
    - **`setColor()`**
    - **`draw()`** ← Depends on the shape!
    - **`getArea()`** ← Might be undefined!

Inheritance for Specification (cont.)

- **in UML :**



Note common (non-UML) notation

<<abstract>>
Shape
- Point location
- Color color
getLocation()
getColor()
setLocation()
abstract draw()

<<abstract>>
ClosedShape
- boolean filled
abstract contains()
abstract getArea()

<<abstract>>
OpenShape
abstract draw()

Circle
- double radius
draw()
getArea()
contains()

Rectangle
- int width, height
draw()
getArea()
contains()

LineSegment
- Point p1, p2
draw()

BezierCurve
- Point p1,p2,p3,p4
draw()

Java Abstract Classes

- Both classes and methods can be declared abstract

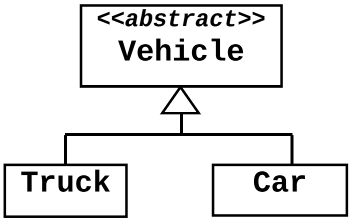
```
public abstract class Animal {
     public abstract void move () ;
}
```

- Abstract classes cannot be instantiated
  - But they can be extended
- If a class contains an abstract method, the class must be declared abstract
  - But abstract classes can also contain concrete methods
- For a subclass to be concrete, it must implement bodies for all inherited abstract methods
  - Otherwise, the subclass is also automatically abstract (and must be declared as such)

Abstract Classes (cont.)

- Can *declare* a variable of abstract type
- Cannot *instantiate* such a variable

```
Vehicle v ;
Truck t = new Truck();
Car c = new Car();
...
v = t ;
...
v = c ;
```



Abstract Classes (cont.)

- **`static`**, **`final`**, and/or **`private`** methods *cannot* be declared abstract
  - No way to override or change them; no way to provide a "specification"

- **`protected`** methods *can* be declared abstract.

- Java "abstract method" = C++ "pure virtual function":

```
    abstract void move () ;        //Java
```

vs.

```
    virtual void move() = 0 ;      //C++
```

Example: Abstract Shapes

```
/** This class is the abstract superclass of all "Shapes". Every Shape has a
 * color, a "location" (origin), accessors, and a draw() method.  */
public abstract class Shape {
  private Color color;
  private Point location;
  public Shape() {
    color = new Color(0,0,0);
    location = new Point (0,0);
  }
  public Point getLocation() {
    return location;
  }
  public Color getColor() {
    return color;
  }
  public void setLocation (Point newLoc) {
    location = newLoc;
  }
  public void setColor (Color newColor) {
    color = newColor;
  }
  public abstract void draw(Graphics g);
}
```

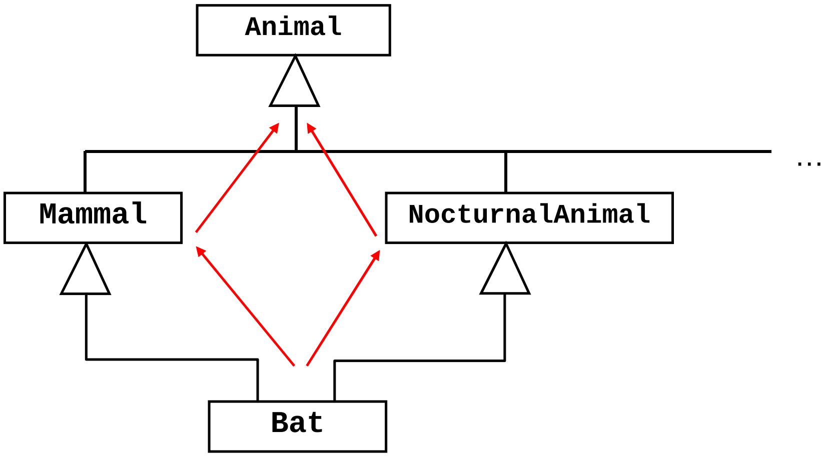
Example: Abstract Shapes (cont.)

```
/** This class defines Shapes which are “closed” – meaning the Shape has a
 *  boundary which delineates “inside” from “outside”.  Closed Shapes can either be
 *  “filled” (solid) or “not filled”  (interior is empty). Every ClosedShape must
 *  have a method “contains(Point)”, which determines whether a given Point is inside
 *  the shape or not, and a method “getArea()” which returns the area inside the shape.
 */
public abstract class ClosedShape extends Shape {
  private boolean filled;          // attribute common to all closed shapes
  public ClosedShape() {
    super();
    filled = false;
  }
  public ClosedShape(boolean filled) {
    this();
    this.filled = filled;
  }
  public boolean isFilled() {
    return filled;
  }
  public void setIsFilled(boolean filled) {
    this.filled = filled;
  }
  public abstract boolean contains(Point p);
  public abstract double getArea();
}
```

Example: Abstract Shapes (cont.)

```
/** This class defines closed shapes which are rectangles. */
public class Rectangle extends ClosedShape {
  private int width;
  private int height;
  public Rectangle() {
    super(true);
    width = 2;
    height = 1;
  }
  public boolean contains(Point p) {
    //... code here to return true if p lies inside this rectangle,
    //    or return false if not.
  }
  public double getArea() {
    return (double) (width * height) ;
  }
  public void draw (Graphics g) {
    if (isFilled()) {
      //  code here to draw a filled (solid) rectangle using
      //  Graphics object "g"
    } else {
      //  code here to draw an empty rectangle using
      //  Graphics object "g"
    }
  }
}
```

Multiple Inheritance



A possible alternative Animal Hierarchy

Multiple Inheritance (cont.)

- C++ allows multiple inheritance:

```
class Animal{};

class Mammal : Animal {
  public : void sleep() {} ;
};

class NocturnalAnimal : Animal {
  public : void sleep() {} ;
};

class Bat : Mammal, NocturnalAnimal { };
```

- Programmer must disambiguate references:

```
void main (int argc, char** argv) {
  Bat aBat ;
  aBat.NocturnalAnimal::sleep();
}
```

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

V – Polymorphism

- **Definition**
- **Static Polymorphism**
- **Dynamic Polymorphism**
- **Polymorphic References**
- **Upcasting / Downcasting**
- **Polymorphic Safety**
- **Java vs. C++**

CSc 133 Lecture Notes

V - Polymorphism

Computer Science Dept.
CSUS

Overview

- Definitions
- Static (“compile-time”) Polymorphism
- Runtime (“dynamic”) Polymorphism
- Polymorphic Safety
- Polymorphism - Java vs. C++

Polymorphism Defined

- Literally: from the Greek

  poly *("many")* + ***morphos*** *("forms")*

- Examples in nature:
  - Carbon: graphite or diamond
  - H_2O: water, ice, or steam
  - Honeybees: queen, drone, or worker

- Programming examples:
  - An operation that can be done in a variety of ways
  - An operation that can be done on various types of objects

"Static" Polymorphism

Detectable *during compilation.*

Example: Operator overloading:

```
  int1 =  int2 + int3 ;
float1 =  float2 + float3 ;
```

- The "+" can perform two *different* operations
- "+" can therefore be thought of as a "*polymorphic operator*"

“Static” Polymorphism (cont.)

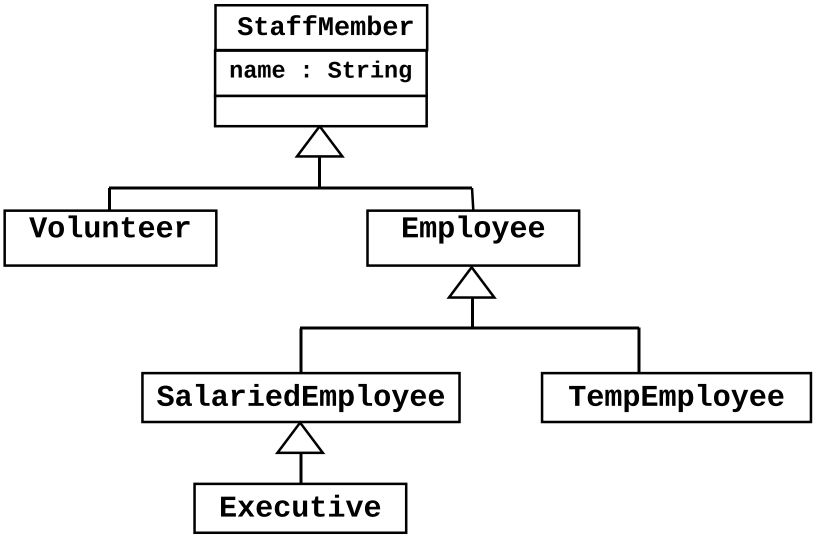
Another example: Method overloading:

```
//return the distance to a location
double distance (int x, int y) { . . . }

//return the distance between two points
double distance (Point p1, Point p2) { . . . }
```

- Same method name, for two different operations
- **“`distance`”** can therefore be thought of as a “*polymorphic method*”

Polymorphic References

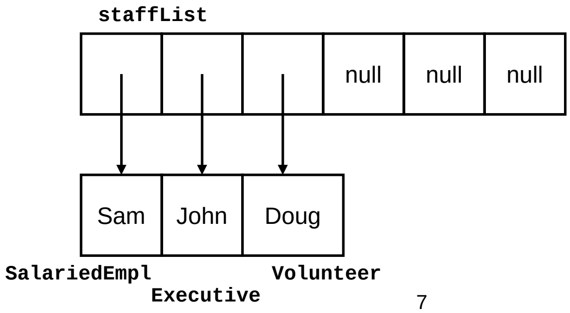
Consider the following class hierarchy:



Polymorphic References (cont.)

- A “polymorphic reference” can
  refer to different object types at runtime:

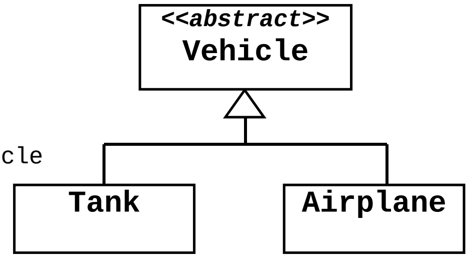
```
StaffMember [ ] staffList = new StaffMember[6];
. . .
staffList[0] = new SalariedEmployee (“Sam”);
staffList[1] = new Executive (“John”);
staffList[2] = new Volunteer (“Doug”);
. . .
```



Upcasting and Downcasting

- “Upcasting” allowed in assignments:

```
Vehicle v ;
Airplane a = new Airplane();
Tank t = new Tank();
...
v = t ;   // a tank IS-A Vehicle
v = a;    // an airplane IS-A Vehicle
```

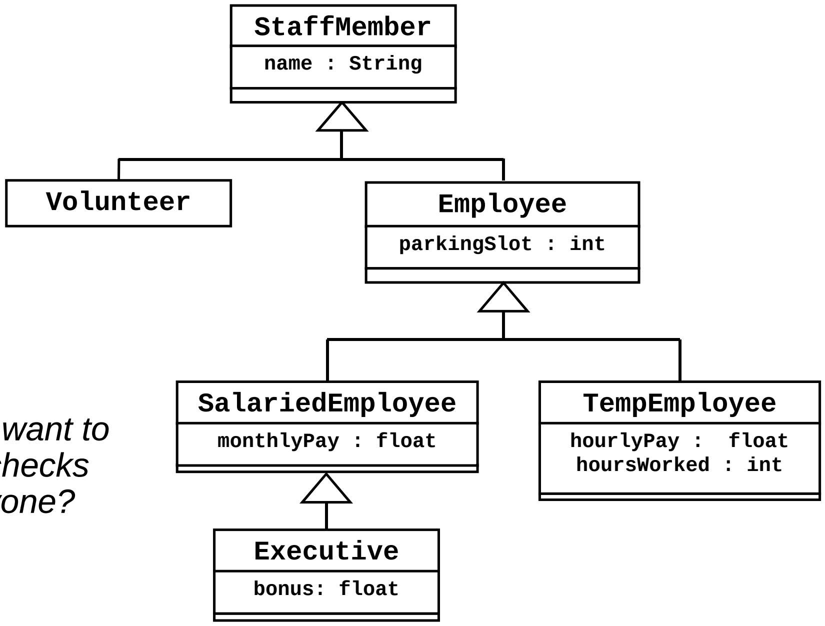


- “Downcasting” requires casting:

```
t = v ;          // compiler error - a Vehicle isn’t a Tank
t = (Tank) v ;  // legal, but dangerous
```

Runtime Polymorphism

Consider this expanded version of the hierarchy shown earlier:



What if we want to print paychecks for everyone?

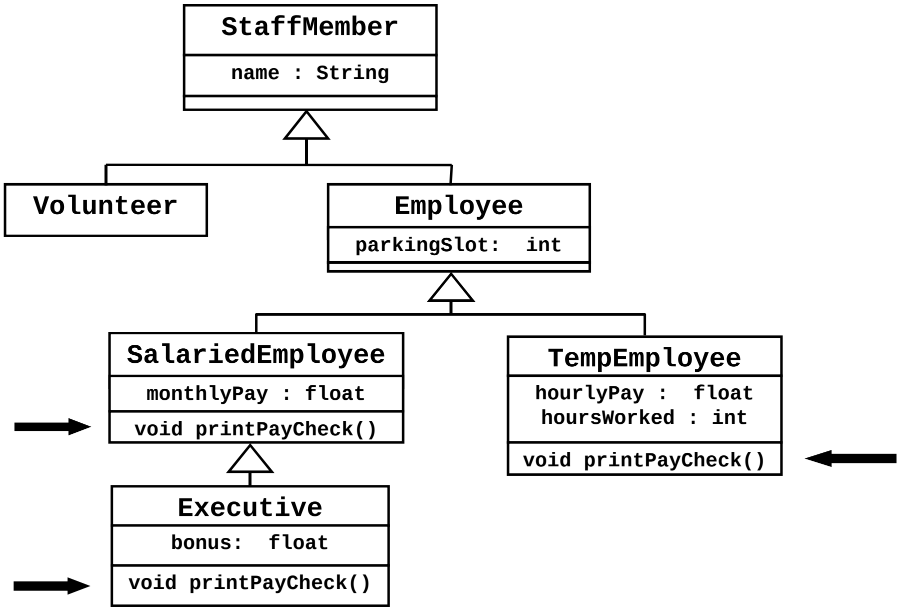
Runtime Polymorphism (cont.)

Printing Paychecks (traditional approach) :

```
for (int i=0; i<staffList.length; i++) {
  String name = staffList[i].getName();
  float amount = 0;
  if (staffList[i] instanceof SalariedEmployee) {
     SalariedEmployee curEmp = (SalariedEmployee) staffList[i];
     amount = curEmp.getMonthlyPay();
     printPayCheck (name, amount);
  } else if (staffList[i] instanceof Executive) {
     Executive curExec = (Executive) staffList[i] ;
     amount = curExec.getMonthlyPay() + curExec.getBonus());
     printPayCheck (name, amount);
  } else if (staffList[i] instanceof TempEmployee) {
     TempEmployee curTemp = (TempEmployee) staffList[i] ;
     amount = curTemp.getHoursWorked()*curTemp.getHourlyPay();
     printPayCheck (name, amount);
  }
}
. . .
private void printPayCheck (String name, float amt) {
 System.out.println (“Pay To The Order Of:” + name + “ $” + amt);
}
```

Runtime Polymorphism (cont.)

First, paycheck computation should be "encapsulated":



Runtime Polymorphism (cont.)

Polymorphic solution:

```
...
for (int i=0; i<staffList.length; i++) {
     staffList[i].printPayCheck() ;
}
...
```

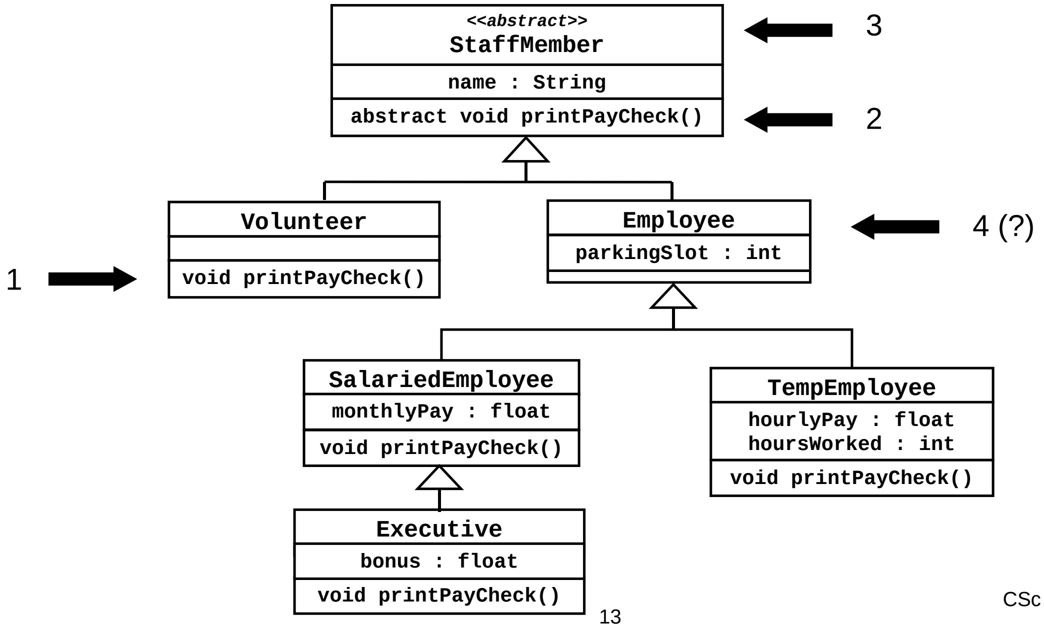
Now, the Print method which gets invoked is:

- *determined at runtime, and*
- *depends on subtype*

We still need to make sure it will compile, and that it is maintainable and extendable…

Polymorphic Safety

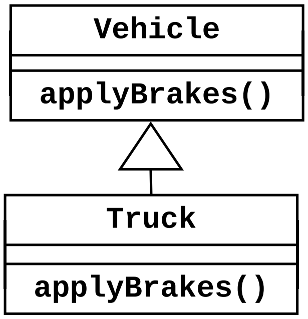
Ideally, *every* class should know how to deal with "**`printPayCheck`**" messages:



Polymorphism: Java vs. C++

- **Java**
  - Run-time (dynamic; late) binding is the default
    - Drawback: may be unnecessary (hence inefficient)
    - Programmer can force compile-time binding by declaring methods (or an entire class) "**`final`**"
- **C++**
  - Compile-time (static; early) binding is the default
    - Drawback: may be inappropriate, since it defaults to calling base-class methods in certain circumstances
    - Programmer can force late binding by declaring methods "**`virtual`**"

Java vs. C++ : Example



C++

```
class Vehicle {
  public:
    void applyBrakes() {
      printf ("Applying vehicle brakes...\n");
    }
};
class Truck : public Vehicle {
  public:
    void applyBrakes() {
      printf ("Applying truck brakes...\n");
    }
};
```

Java

```
class Vehicle {
  public void applyBrakes() {
    System.out.printf ("Applying vehicle brakes\n");
  }
}
class Truck extends Vehicle {
  public void applyBrakes() {
    System.out.printf("Applying truck brakes...\n");
  }
}
```

Java vs. C++ : Example (cont.)

C++

```
void main (int argc, char** argv){
  Vehicle * pV ;
  Truck * pT ;
  pT = new Truck();
  pT->applyBrakes();
  pV = pT;
  pV->applyBrakes();
}
```

Java

```
public static void main (String [] args){
  Vehicle v;
  Truck t;
  t = new Truck();
  t.applyBrakes();
  v = t ;
  v.applyBrakes();
}
```

Output

```
Applying truck brakes...
Applying vehicle brakes...
```

```
Applying truck brakes...
Applying truck brakes...
```

Run C++

Run Java

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

VI – Interfaces

- **Class Interfaces**
- **Interface Design**
- **The Java *Interface* Construct**
- **Interfaces in C++**
- **Interface Subtypes**
- **UML notation**
- **Predefined interfaces**
- **Interface hierarchies**
- **Interfaces and polymorphism**
- **Abstract Classes vs. Interfaces**
- **Multiple Inheritance**

CSc 133 Lecture Notes

VI - Interfaces

Computer Science Dept.
CSUS

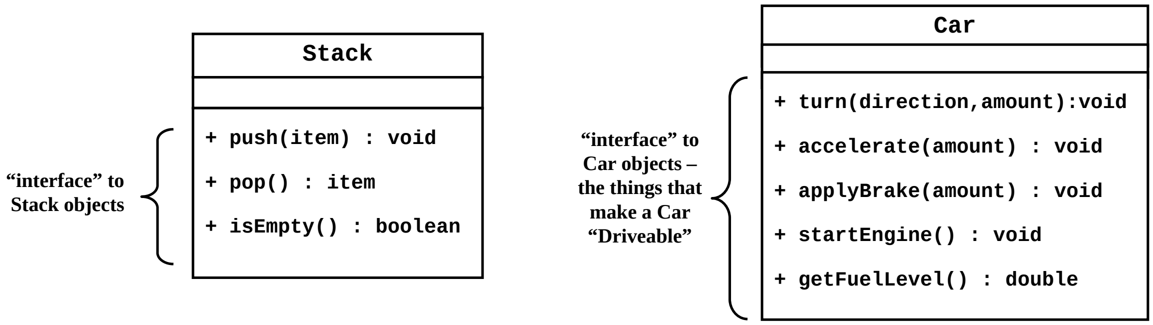
Overview

- Class Interfaces
- The Java *Interface* Construct
- Interfaces in C++
- Interface Subtypes
- Interfaces and Polymorphism
- Multiple Inheritance via Interfaces

CLASS INTERFACES

Every class definition creates an "interface"

- The exposed (non-private) parts of an object



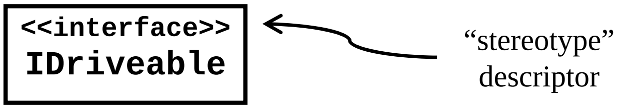
"interface" to Stack objects

```
Stack

+ push(item) : void
+ pop() : item
+ isEmpty() : boolean
```

"interface" to Car objects – the things that make a Car "Driveable"

```
Car

+ turn(direction,amount):void
+ accelerate(amount) : void
+ applyBrake(amount) : void
+ startEngine() : void
+ getFuelLevel() : double
```

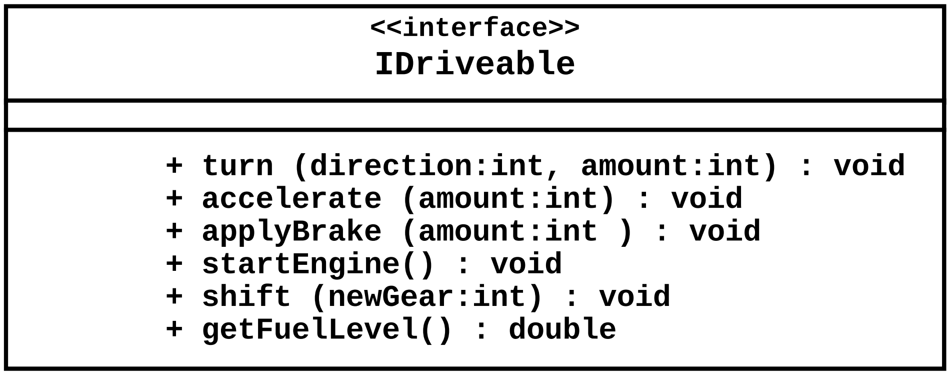
UML Interface Notation



```
<<interface>>
IDriveable
```

"stereotype" descriptor

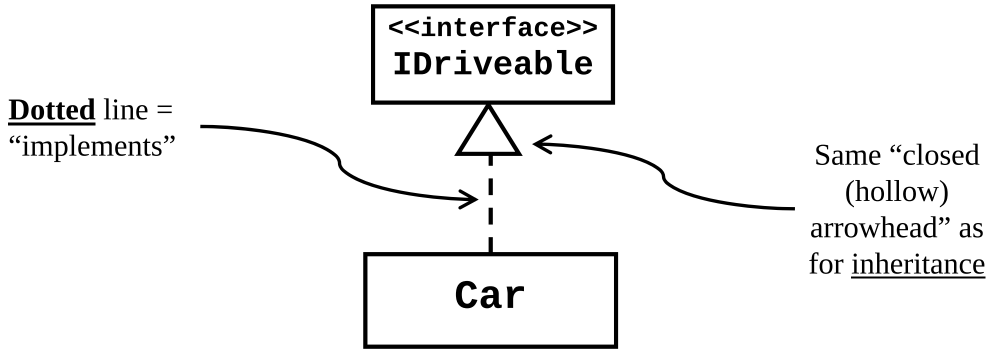
or



```
<<interface>>
IDriveable

+ turn (direction:int, amount:int) : void
+ accelerate (amount:int) : void
+ applyBrake (amount:int ) : void
+ startEngine() : void
+ shift (newGear:int) : void
+ getFuelLevel() : double
```

UML Interface Notation (cont.)

- Class **`Car`** implements interface "**`IDriveable`**":

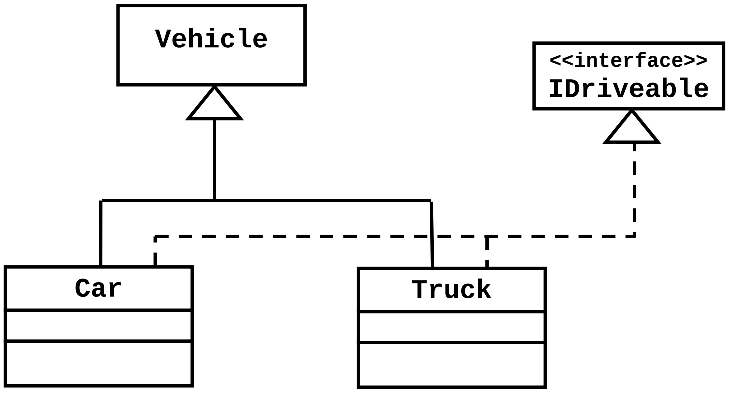


Dotted line = "implements"

Same "closed (hollow) arrowhead" as for inheritance

UML Interface Notation (cont.)

- **Car and Truck both derive from "`Vehicle`"**
- **Car and Truck both implement "`IDriveable`"**



Java *Interface* construct

Characteristics of a class “interface” :

- Defines a set of methods with specific signatures
  - Methods are *implicitly* **public** and **abstract**
- Does *not* specify the *implementation* of the methods

Java allows specification of an “interface” independently from any particular class:

```
public interface IDriveable {
   void turn (int direction, int amount);
   void accelerate (int amount);
   void applyBrake (int amount);
   void startEngine ( );
   void shift (int newGear);
   double getFuelLevel ( );
}
```

Using Java Interfaces

Classes can agree to “implement” an interface:

```
public class Car extends Vehicle implements IDriveable {
    public void turn (int direction, int amount) {...}
    public void accelerate (int amount) {...}
    public void applyBrake (int amount) {...}
    public void startEngine() {...}
    public void shift (int newGear) {...}
    public double getFuelLevel ( ) {...}
    /*... other Car methods (if any) here ... */
}
```

- “`implements`” == “provides bodies for all methods”
- *Compiler checks!*

Using Java Interfaces (cont.)

Multiple classes may provide the same *interface* but with different *implementations*

- **Example: Truck also implements "`IDriveable`" – but in a different way:**

```
public class Truck extends Vehicle implements IDriveable {
    public void turn (int direction, int amount) {...}
    public void accelerate (int amount) {...}
    public void applyBrake (int amount)
      { different code here to apply Truck brakes... }
    public void startEngine()
      { truck engine startup code... }
    public void shift (int newGear)
      { truck shifting code... }
    public double getFuelLevel ( )
      { code to check multiple fuel tanks... }
    /*... other Truck methods here ... */
}
```

Interface Inheritance

- Subclasses *inherit* interface implementations

```
public interface IDriveable {
  void turn (int dir, int amt);
  void accelerate (int amt);
  void applyBrake (int amt);
  void startEngine ( );
  void shift (int newGear);
  double getFuelLevel ( );
}
```

```
public class Vehicle implements IDriveable {
  public void turn(int dir, int amt){...}
  public void accelerate (int amt) {...}
  public void applyBrake (int amt) {...}
  public void startEngine( ) {...}
  public void shift (int newGear) {...}
  public double getFuelLevel ( ) {...}
}
```

```
public class Car extends Vehicle {
  public void applyBrake (int amt) {...}
  public void startEngine ( ) {...}
  public void shift (int newGear) {...}
  public double getFuelLevel( ) {...}
  // Car doesn't need to specify "turn()" or "accelerate()"
  // because they are inherited from Vehicle
}
```

“Interfaces” In C++

- **“Abstract” Methods:**

```
virtual void turn (int direction, int amount) = 0 ;
```

- **“Abstract” Classes :**

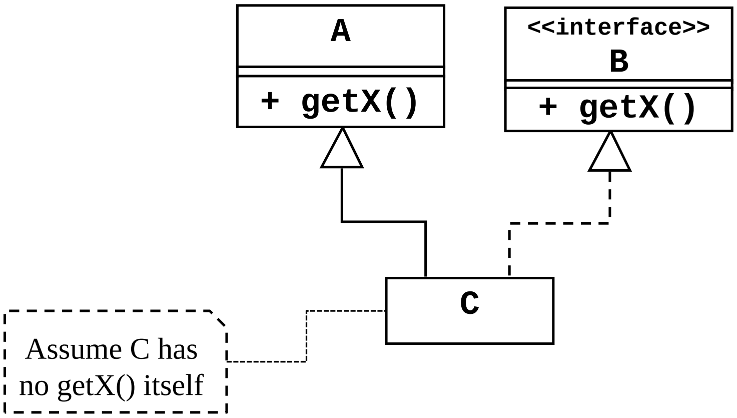
```
class IDriveable {
  public:
        virtual void turn (int direction, int amount) = 0 ;
        virtual void accelerate (int amount) = 0 ;
        ...
};
```

- **“Abstract” Classes as Interfaces :**

```
class Vehicle { ... };
class Car : public IDriveable, Vehicle
{ ... };
```

Quiz:

Which **getX()** is called in objects of type C?



Assume C has no getX() itself

Predefined **Interfaces In Java**

- **Many Java Classes implement built-in interfaces**
- **User Classes can also implement them**

Examples:

```
interface Shape {
  boolean contains (Point2D point);
  boolean intersects (Rectangle2D rect);
  Rectangle2D getBounds ( );
  //other methods...
}
```

```
interface Comparable {
  int compareTo (Object otherObj);
}
```

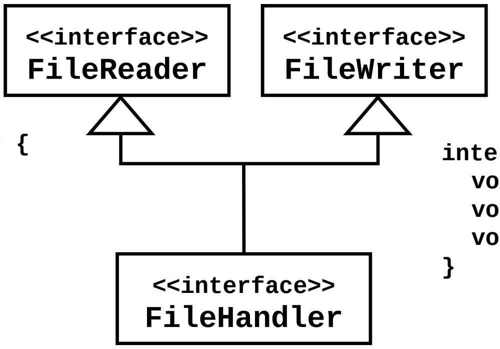
```
interface AudioClip {
    void loop ( );
    void play ( );
    void stop ( );
}
```

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

Interface Hierarchies

Interfaces can *extend* other interfaces

```
interface FileReader {
  byte readByte();
  int readInt();
  String readLine();
}
```



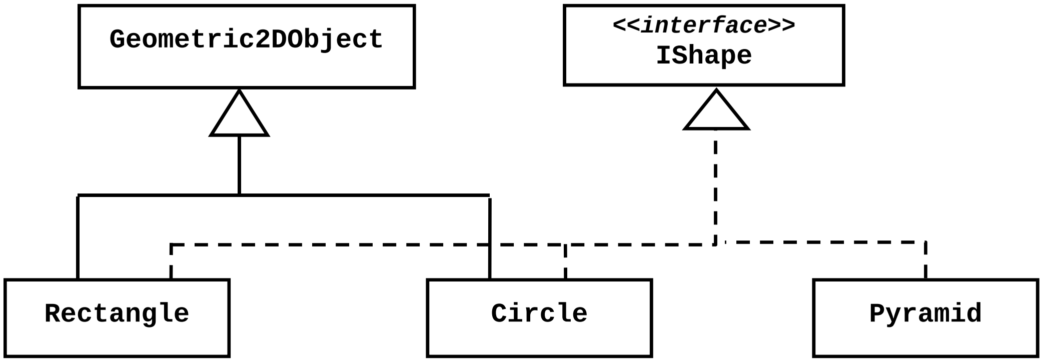
```
interface FileWriter {
  void writeByte (byte b);
  void writeInt (int theInt);
  void writeString (String s);
}
```

```
interface FileHandler extends FileReader, FileWriter {
  void open (String filename);
  void close ( );
}
```

Interface Subtypes

If a Class implements an Interface, it is considered a "subtype" of the "interface type":

- A Circle "IS-A" Geometric2DObject
- A Circle "IS-A" Shape



Interface Subtypes (cont.)

- **Objects can be upcast to** *interface types:*

```
Circle myCircle = new Circle();
IShape myShape = (IShape) myCircle ;
```

- **Interfaces provide objects with:**

  "apparent type" vs. *"actual type"*

- ***Variable of interface type can hold many different types of objects!***

Interfaces and Polymorphism

- *Apparent* type = the *interface presented*

  "What can be invoked"

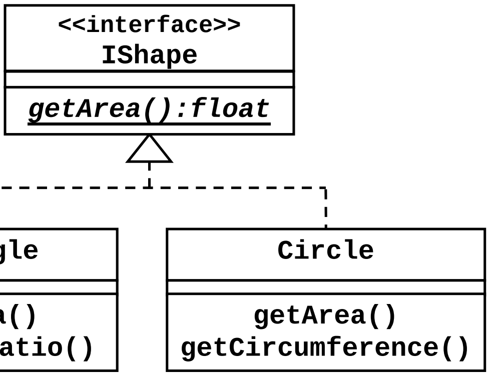
- *Actual* type = the *behavior*

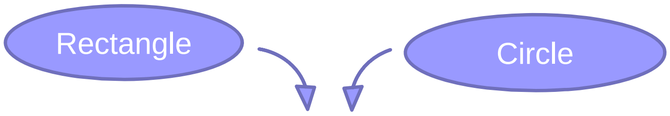
  "What happens when invoked"

```
IShape [ ] myThings = new IShape [10] ;
myThings[0] = new Rectangle();
myThings[1] = new Circle();
//...code here to add more rectangles, circles, or other "shapes"

for (int i=0; i<myThings.length; i++) {
  IShape nextThing = myThings[i];
  process ( nextThing );
}
...
void process (IShape aShape) {
  // code here to process a Shape object, making calls to Shape methods.
  // Note this code only knows the apparent type, and only Shape methods
  // are visible – but any methods invoked are those of the actual type.
}
```

Interface Polymorphism Example

- Suppose we have:

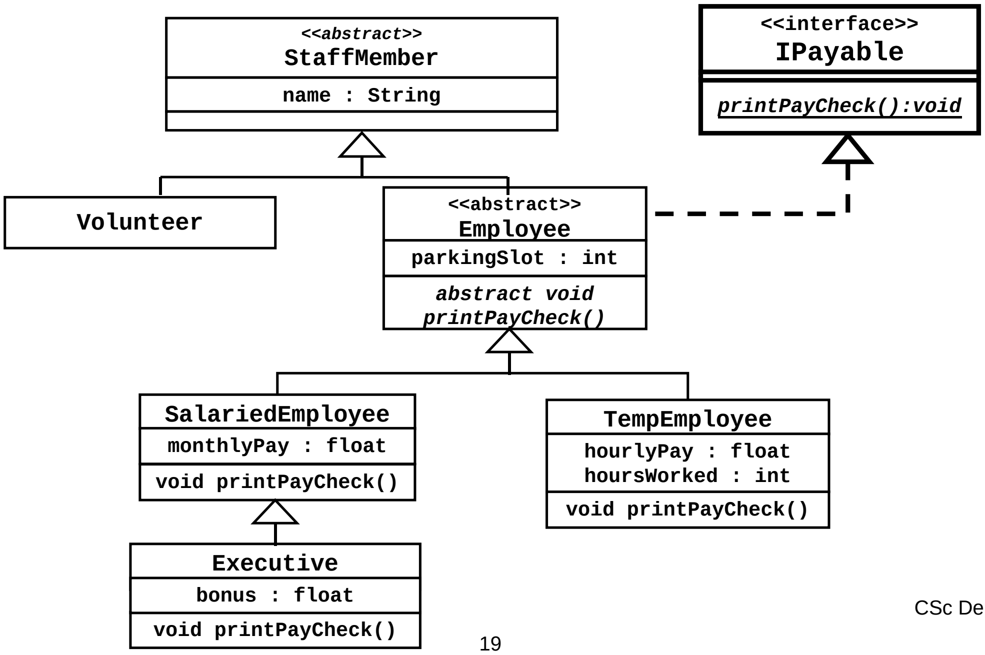




```
void process (IShape s) {
  ...
  s.getArea();            //legal; all IShapes have getArea()
  ...
  s.getAspectRatio();     //illegal,  even if 's' is a Rectangle
                          //  (generates a compiler error)
}
```

Polymorphic Safety Revisited

- StaffMember hierarchy using Interfaces:



Interface Polymorphic Safety

```
public class StaffMember {
   ...
}

public interface IPayable {
   public void printPayCheck() ;
}

//Every kind of ”Employee” IS-A ”payable” (must provide printPayCheck())
public class Employee extends StaffMember implements IPayable {
   ...
   abstract public void printPayCheck() ;
}
```

```
//client using interface polymorphism to safely print paychecks:
for (int i=0; i<staffList.length; i++) {
   if (staffList[i] instanceof IPayable)  ⬅
      ((IPayable)staffList[i]).printPayCheck() ;
}
```

Abstract Classes vs. Interfaces

```
abstract class Animal {
  abstract void talk();
}

class Dog extends Animal {
  void talk() {
    System.out.println(“Woof!”);
  }
}

class Cat extends Animal {
  void talk() {
    System.out.println(“Meow!”);
  }
}
```

```
class Example {
  ...
  Animal animal = new Dog();
  Interrogator.makeItTalk(animal);
  animal = new Cat();
  Interrogator.makeItTalk(animal);
  ...
}
```

```
class Interrogator {
 static void
    makeItTalk(Animal subject) {
      subject.talk();
    }
}
```

Example after Bill Venners, www.javaworld.com

Abstract Classes vs. Interfaces (cont.)

- We can easily add a Bird and “make it talk”:

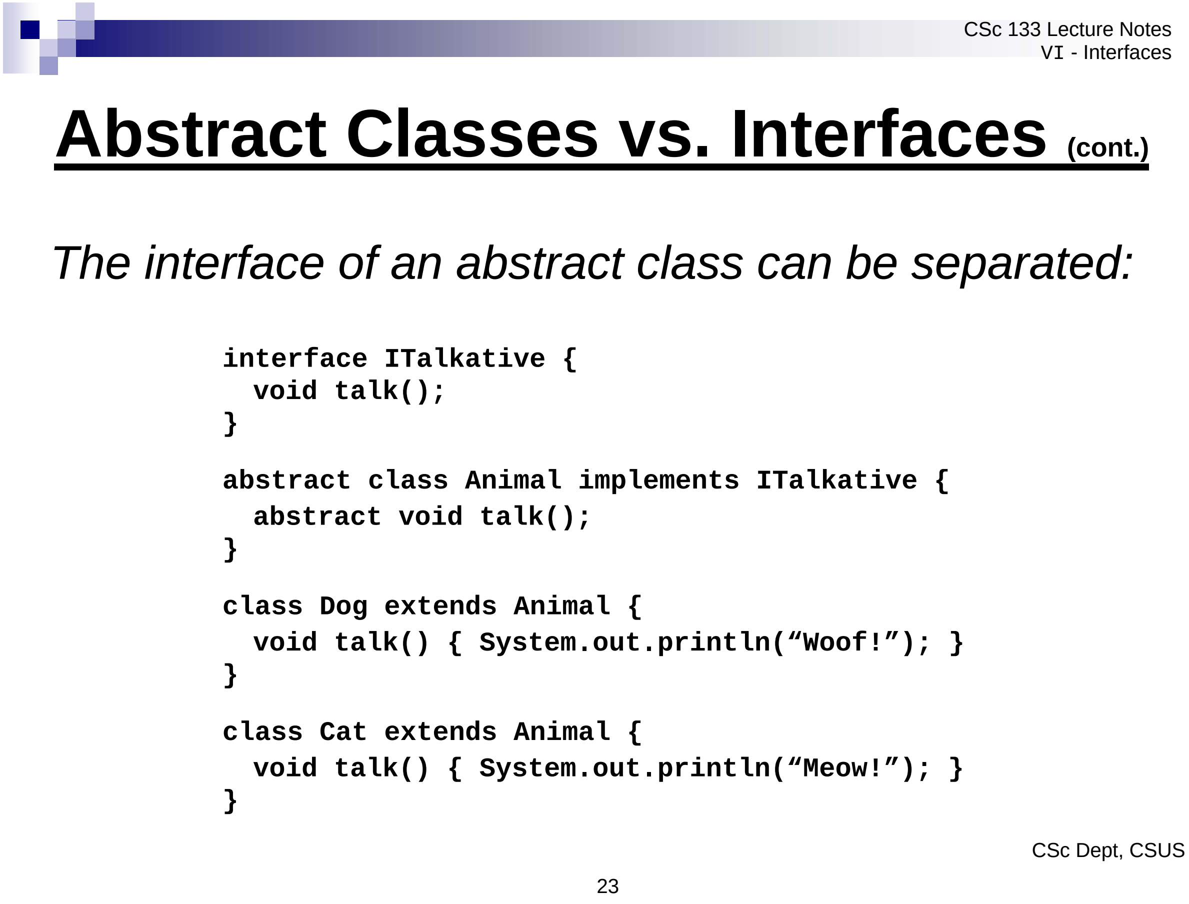
```
class Bird extends Animal {
  void talk() {
    System.out.println(“Tweet! Tweet!”);
  }
}
```

- Making a CuckooClock “talk” is a problem:

```
class Clock {... }
class CuckooClock extends Clock {
  void talk() {
    System.out.println(“Cuckoo! Cuckoo!”);
  }
}
```

We can’t pass a CuckooClock to Interrogator – it’s not an animal.

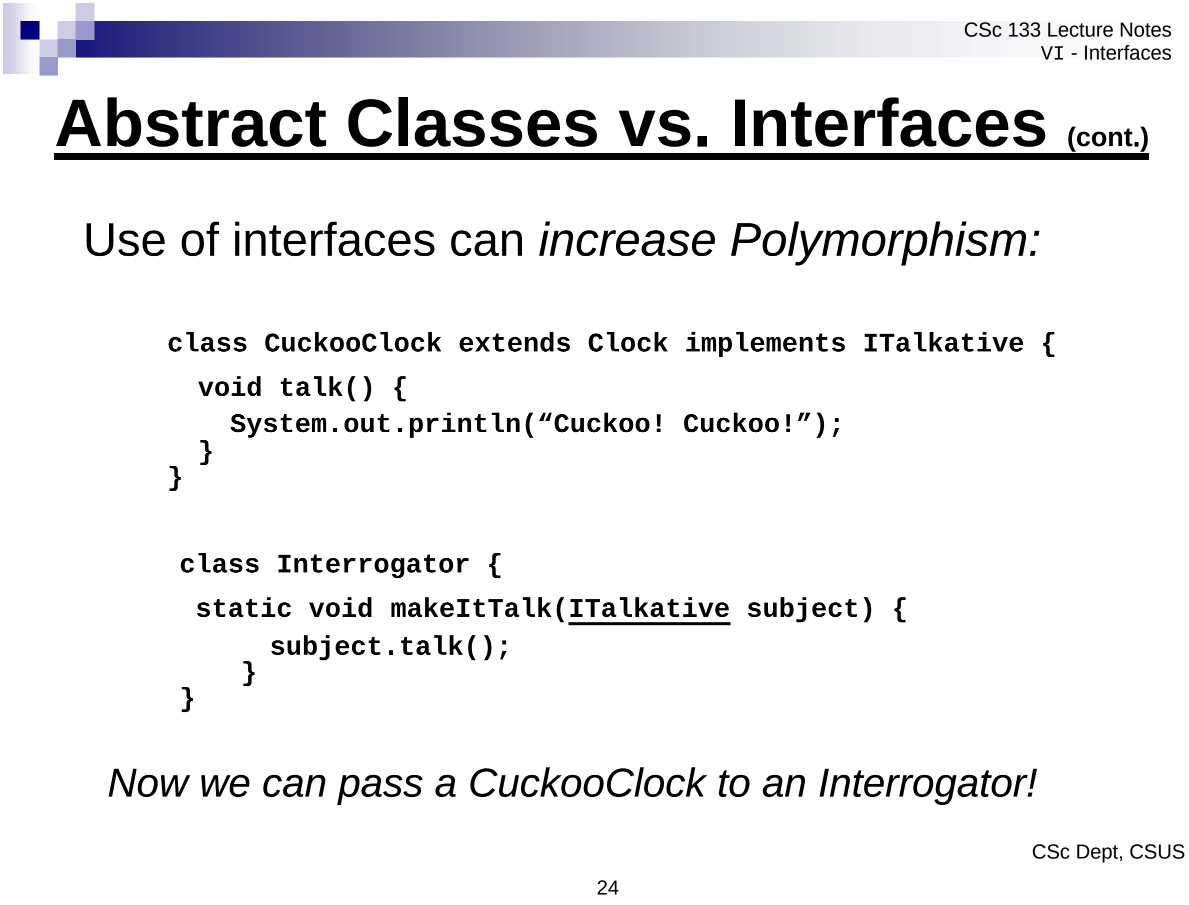
And it is <u>illegal</u> (in Java) to <u>also</u> extend animal (can only “extend” once!)



Abstract Classes vs. Interfaces (cont.)

The interface of an abstract class can be separated:

```
interface ITalkative {
  void talk();
}

abstract class Animal implements ITalkative {
  abstract void talk();
}

class Dog extends Animal {
  void talk() { System.out.println(“Woof!”); }
}

class Cat extends Animal {
  void talk() { System.out.println(“Meow!”); }
}
```



Abstract Classes vs. Interfaces (cont.)

Use of interfaces can *increase Polymorphism:*

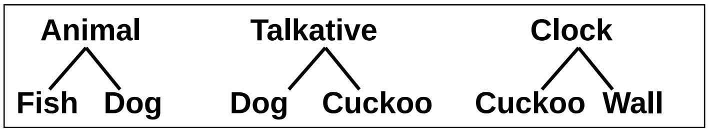
```
class CuckooClock extends Clock implements ITalkative {
  void talk() {
    System.out.println(“Cuckoo! Cuckoo!”);
  }
}
```

```
class Interrogator {
 static void makeItTalk(ITalkative subject) {
      subject.talk();
    }
}
```

Now we can pass a CuckooClock to an Interrogator!

Abstract Classes vs. Interfaces (cont.)

Interfaces allow for *multiple hierarchies*:



```
interface ITalkative {
  void talk();
}

abstract class Animal {
  abstract void move();
}

class Fish extends Animal { // not talkative!
  void move() { //code here for swimming }
}

class Dog extends Animal implements ITalkative {
  void talk() { System.out.println("Woof!"); }
  void move() { //code here for walking/running }
}

class CuckooClock extends Clock implements ITalkative {
  void talk() { System.out.println("Cuckoo!"); }
}
```

Abstract Class vs. Interface: Which?

Abstract classes are a good choice when:

- There is a clear *inheritance hierarchy* to be defined (e.g. "kinds of animals")
- There are at least *some* concrete methods shared between subtypes (e.g. *getName()*)
- There may be a need to *add new methods* in the future
  - Adding methods to an *interface* breaks all implementing classes (not true of adding methods to an abstract class)

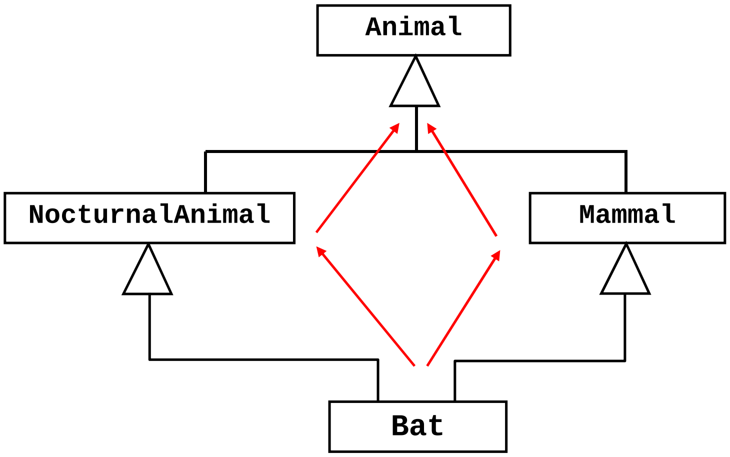
Abstract Class vs. Interface: Which?

Interfaces are a good choice when:

- The relationship between the methods and the implementing class is not extremely strong
  - Example: many classes implement “Comparable” or “Cloneable”; these concepts are not tied to a specific class
- An API is likely to be stable
  - Again: adding interface methods *breaks implementing classes*
- Something like Multiple Inheritance is desired

(see next slides…)

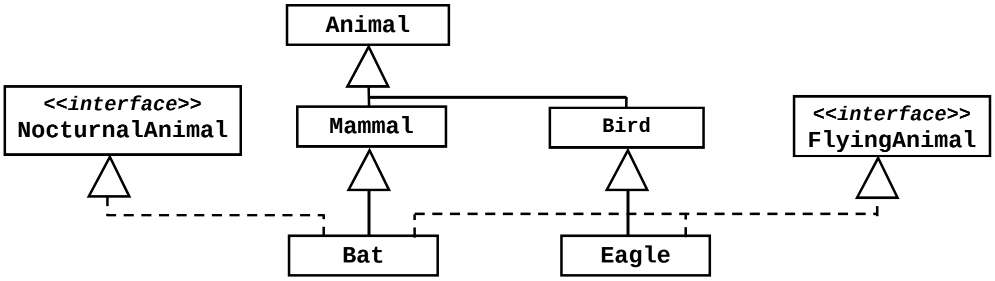
Multiple Inheritance Revisited



A possible alternative Animal Hierarchy

Multiple Inheritance via *Interfaces*

Can say this exactly in Java:



```
public class Animal {...}
public class Mammal extends Animal {...}
public interface NocturnalAnimal {...}
public class Bat extends Mammal implements NocturnalAnimal {...}
```

and more:

```
public interface FlyingAnimal {...}
public class Bat extends Mammal implements NocturnalAnimal,
                          FlyingAnimal  {...}
```

Does Java support multiple inheritance?

- Of interfaces – Yes
- Of implementations – No

<< This page intentionally left (almost) blank >>

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

VII - Design Patterns

- **Background**
- **Categories of Design Patterns**
- **Specific Patterns:**
  - ***Iterator***
  - ***Composite***
  - ***Singleton***
  - ***Observer***
  - ***Adapter***
  - ***Command***
  - ***Decorator***
  - ***Factory Method***
  - ***Memento***
  - ***Proxy***
  - ***Strategy***
- **MVC Architecture**

(note – the *Façade* Pattern is covered in Chapter 17)

CSc 133 Lecture Notes

VII - Design Patterns

Computer Science Dept.
CSUS

Overview

- **Background**
- **Types of Design Patterns**
  - **Creational vs. Structural vs. Behavioral Patterns**
- **Specific Patterns**

Composite
Iterator
Adapter
Decorator
Memento
Strategy
Singleton
Observer
Command
Factory Method
Proxy

- **MVC Architecture**

Background

- A generic, clever, useful, or insightful solution to a set of recurring problems.
- Popularized by 1995 book: ***"Design Patterns: Elements of Reusable Object-Oriented Software"*** by Gamma et. al (the "gang of four").
  … identified the original set of 23 patterns.
- Original concept from architecture:
  door, hallway, alcove, balustrade, etc.
- Code frequently needs to do things that have been done before.

Types of Design Patterns

- **CREATIONAL**
  - Deal with process of object creation
- **STRUCTURAL**
  - Deal with structure of classes – how classes and objects can be combined to form larger structures
  - Design objects that satisfy constraints
  - Specify connections between objects
- **BEHAVIORAL**
  - Deal with interaction between objects
  - Encapsulate processes performed by objects

Common Design Patterns

As defined in *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helm, Johnson, and Vlissides

Creational:

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural:

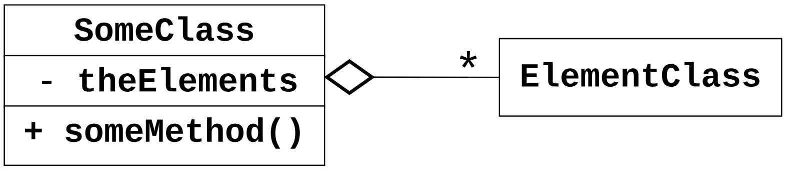
- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral:

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

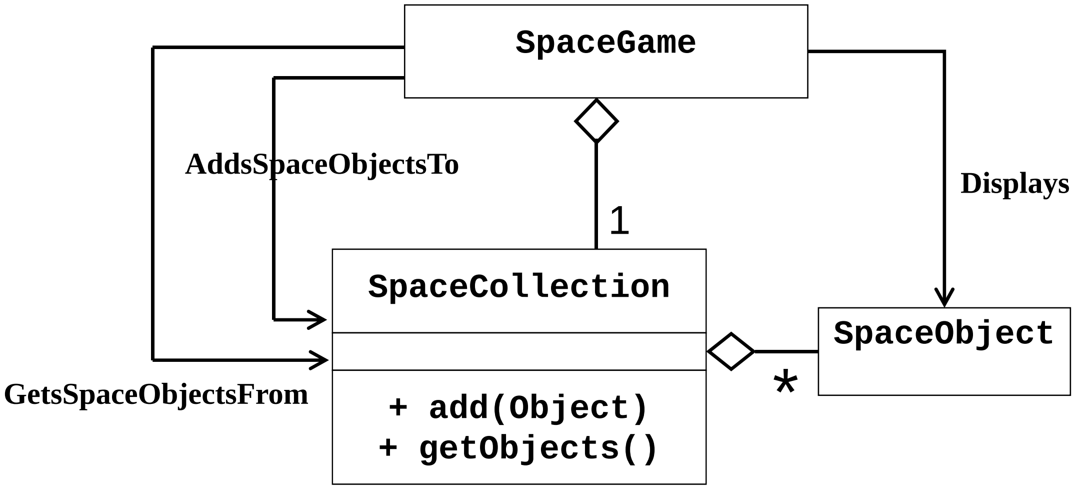
The Iterator Pattern

- MOTIVATION
  - An “aggregate” object contains “elements”
  - “Clients” need to access these elements
  - Aggregate shouldn’t expose internal structure



- Example
  - *GameWorld* has a set of game characters
  - *Screen view* needs to display the characters
  - *Screen view* needn’t know the data structure

Collection Classes



SpaceGame

AddsSpaceObjectsTo

Displays

1

SpaceCollection

+ add(Object)
+ getObjects()

SpaceObject

*

GetsSpaceObjectsFrom

SpaceGame Implementation

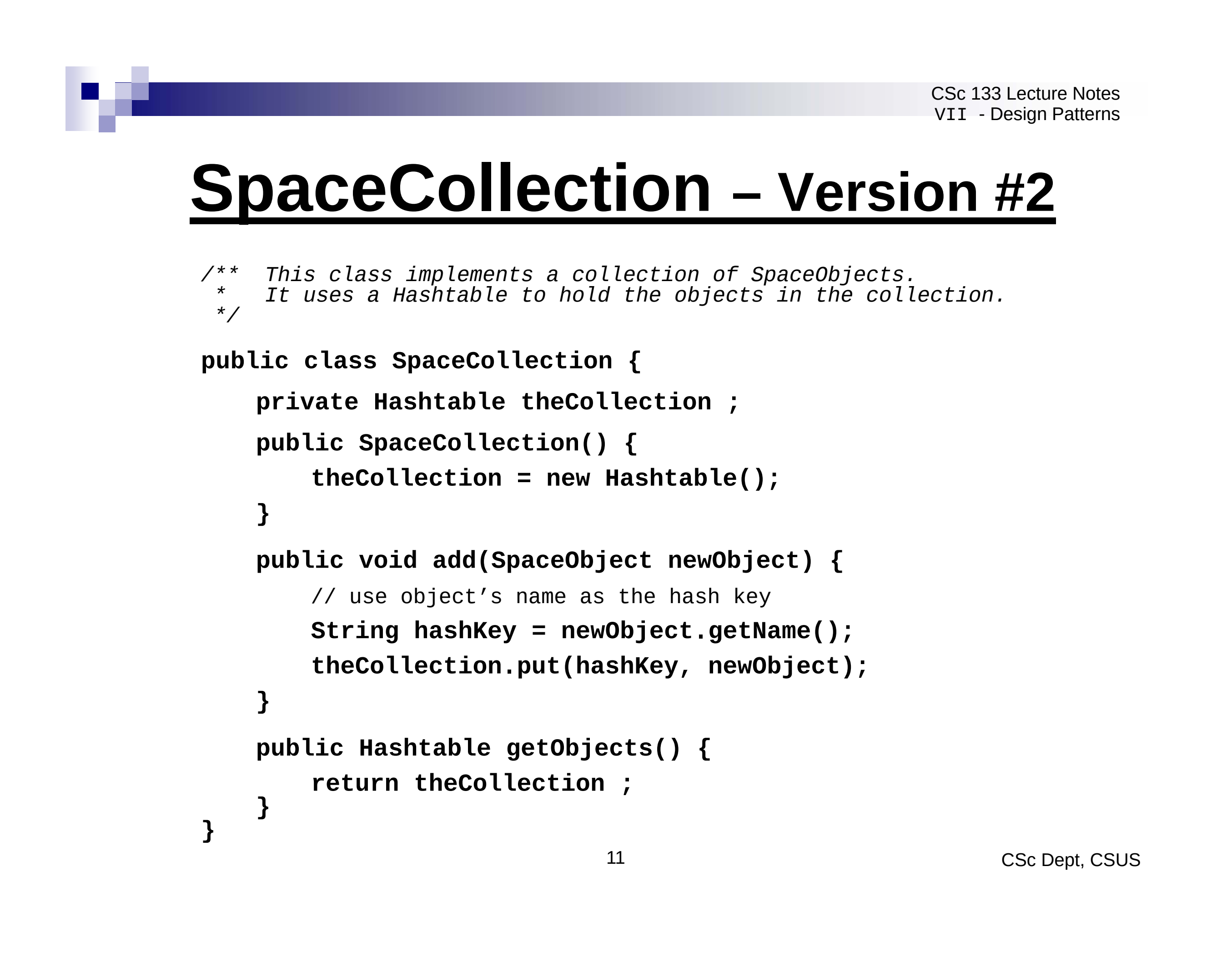
```
import java.util.Vector;
/** This class implements a game containing a collection of SpaceObjects.
 *  The class has knowledge of the underlying structure of the collection
*/
public class SpaceGame {
   private SpaceCollection theSpaceCollection ;
   public SpaceGame() {
       //create the collection
       theSpaceCollection = new SpaceCollection();
       //add some objects to the collection
       theSpaceCollection.add (new SpaceObject(“Obj1”));
       theSpaceCollection.add (new SpaceObject(“Obj2”));
       ...
   }
   //display the objects in the collection
   public void displayCollection() {
       Vector theObjects = theSpaceCollection.getObjects();
       for (int i=0; i<theObjects.size(); i++) {
          System.out.println (theObjects.elementAt(i));
       }
   }
}
```

Space Objects

```
/** This class implements a Space object.
 *  Each SpaceObject has a name and a location.
 */
public class SpaceObject {
    private String name;
    private Point location;
    public SpaceObject (String theName) {
      name = theName;
      location = new Point(0,0);
    }
    public String getName() {
      return name;
    }
    public Point getLocation() {
      return new Point (location);
    }
    public String toString() {
      return "SpaceObject " + name + " " + location.toString();
    }
}
```

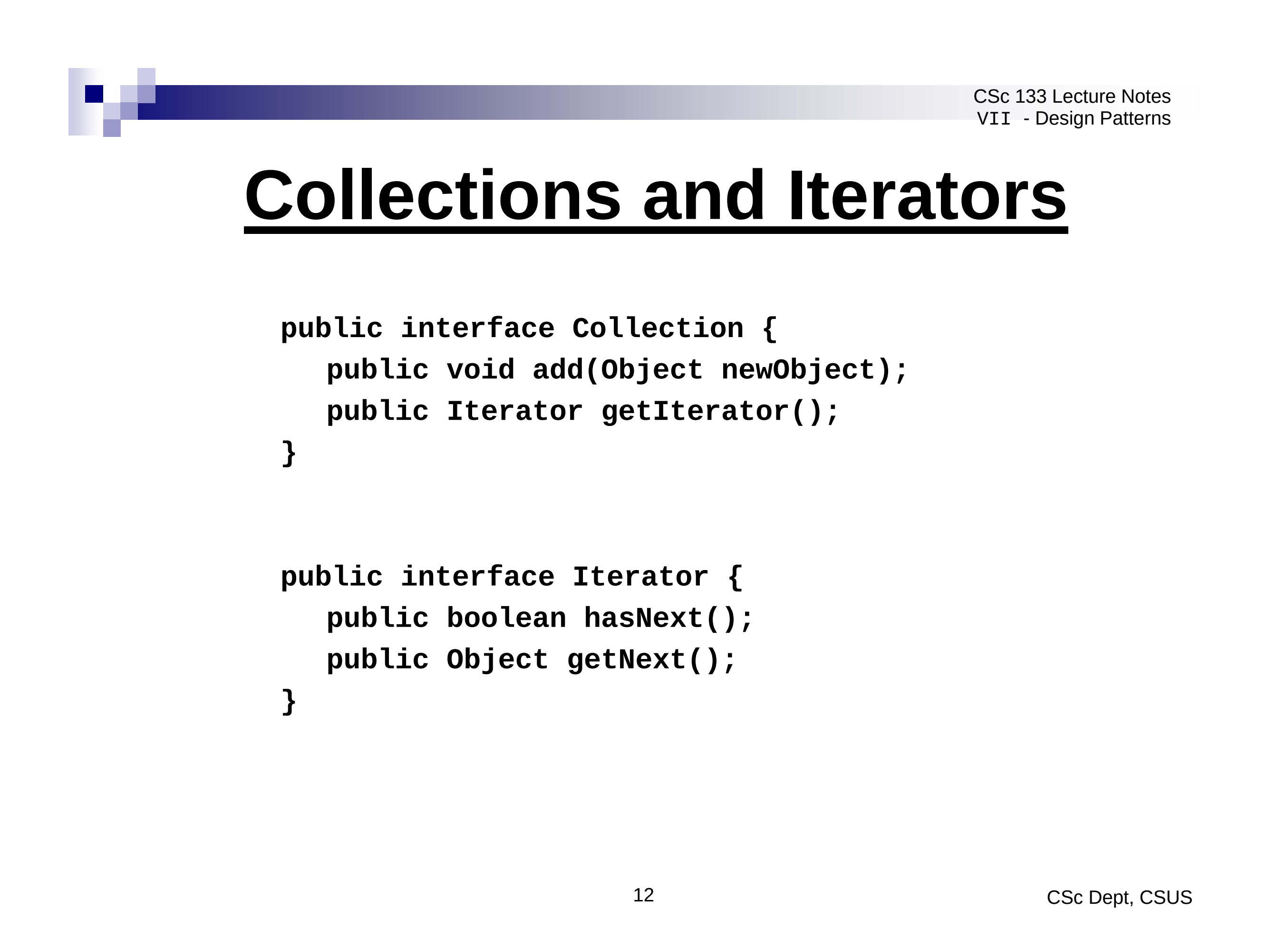
SpaceCollection – Version #1

```
/**  This class implements a collection of SpaceObjects.
 *   It uses a Vector to hold the objects in the collection.
 */
public class SpaceCollection {
    private Vector theCollection ;
    public SpaceCollection() {
        theCollection = new Vector();
    }
    public void add(SpaceObject newObject) {
        theCollection.addElement(newObject);
    }
    public Vector getObjects() {
        return theCollection ;
    }
}
```



SpaceCollection – Version #2

```
/**  This class implements a collection of SpaceObjects.
 *   It uses a Hashtable to hold the objects in the collection.
 */
public class SpaceCollection {
    private Hashtable theCollection ;
    public SpaceCollection() {
        theCollection = new Hashtable();
    }
    public void add(SpaceObject newObject) {
        // use object's name as the hash key
        String hashKey = newObject.getName();
        theCollection.put(hashKey, newObject);
    }
    public Hashtable getObjects() {
        return theCollection ;
    }
}
```



Collections and Iterators

```
public interface Collection {
   public void add(Object newObject);
   public Iterator getIterator();
}


public interface Iterator {
   public boolean hasNext();
   public Object getNext();
}
```

SpaceCollection With Iterator

```
/** This class implements a collection of SpaceObjects.
 *  It uses a Vector as the structure but does
 *  NOT expose the structure to other classes.
 *  It provides an iterator for accessing the
 *  objects in the collection.
 */
public class SpaceCollection implements Collection {

    private Vector theCollection ;

    public SpaceCollection() {
        theCollection = new Vector ( );
    }

    public void add(Object newObject) {
        theCollection.addElement(newObject);
    }

    public Iterator getIterator() {
        return new SpaceVectorIterator ( ) ;
    }

    ...continued...
```

SpaceCollection With Iterator (cont.)

```
  private class SpaceVectorIterator implements Iterator {
      private int currElementIndex;

      public SpaceVectorIterator() {
        currElementIndex = -1;
      }

      public boolean hasNext() {
          if (theCollection.size ( ) <= 0)  return false;
          if (currElementIndex == theCollection.size() – 1 )
            return false;
          return true;
      }

      public Object getNext ( ) {
          currElementIndex ++ ;
          return(theCollection.elementAt(currElementIndex));
      }
  } //end private iterator class

} //end SpaceCollection class
```

Using An Iterator

```
/** This class implements a game containing a collection of SpaceObjects.
 *  The class assumes no knowledge of the underlying structure of the
 *  collection -- it uses an Iterator to access objects in the collection.
 */
public class SpaceGame {
     private SpaceCollection theSpaceCollection ;
     public SpaceGame() {
          //create the collection
          theSpaceCollection = new SpaceCollection();
          //add some objects to the collection
          theSpaceCollection.add (new SpaceObject(“Obj1”));
          theSpaceCollection.add (new SpaceObject(“Obj2”));
          ...
     }
     //display the objects in the collection
     public void displayCollection() {
          Iterator theElements = theSpaceCollection.getIterator() ;
          while ( theElements.hasNext() ) {
               SpaceObject spo = (SpaceObject) theElements.getNext() ;
               System.out.println ( spo ) ;
          }
     }
}
```

Java’s `Iterator` Interface

`boolean hasNext()`

Returns true if the collection has more elements.

`Object next()`

Returns the next element in the collection.

`void remove()`

Removes from the collection the last element returned by the iterator (optional operation).

Java’s Collection Interface

boolean add(Object o) : Ensures that this collection contains the specified element

boolean addAll(Collection c) : Adds all of the elements in the specified collection to this collection

void clear() : Removes all of the elements from this collection

boolean contains(Object o) : Returns true if this collection contains the specified element.

boolean containsAll(Collection c) : Returns true if this collection contains all of the elements in the specified collection.

boolean equals(Object o) : Compares the specified object with this collection for equality.

int hashCode() : Returns the hash code value for this collection.

boolean isEmpty() : Returns true if this collection contains no elements.

Iterator iterator() : Returns an iterator over the elements in this collection.

boolean remove(Object o) : Removes a single instance of the specified element from this collection, if it is present

boolean removeAll(Collection c) : Removes all this collection's elements that are also contained in the specified collection

boolean retainAll(Collection c) : Retains only the elements in this collection that are contained in the specified collection

int size() : Returns the number of elements in this collection.

Object[] toArray() : Returns an array containing all of the elements in this collection.

Object[] toArray(Object[] a) : Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

Java’s Iterable Interface

- **Implementing this interface allows an object to be the target of the “*foreach*” statement…**

```
interface Iterable {
   public Iterator iterator();      // "getIterator()"
}
```

- **Example:**

```
public class SpaceCollection implements Iterable {
   ...
   public Iterator iterator() {
      return new SpaceVectorIterator(); //as before
   }
}
public class SpaceGame {
   ...
   public void displayCollection() {
      for (Object spo : theSpaceCollection){   //"foreach"
         System.out.println (((SpaceObject)spo).getName());
      }
   }
}
```

Iterators In C++

- C++ ***Standard Template Library (STL)*** **provides *container classes* (e.g. vector, map, list, …)**
- **All containers provide *iterators* over their contents, using functions returning pointers:**

```
using namespace std;
vector<SpaceObject> myObjs;    //create a container
  //... code here to add SpaceObjects to the container (vector)
vector<SpaceObject>::iterator itr;    //declare an iterator
for (itr = myObjs.begin(); itr != myObjs.end(); itr++) {
   cout << *itr ;    //output next obj pointed to by itr
}
```

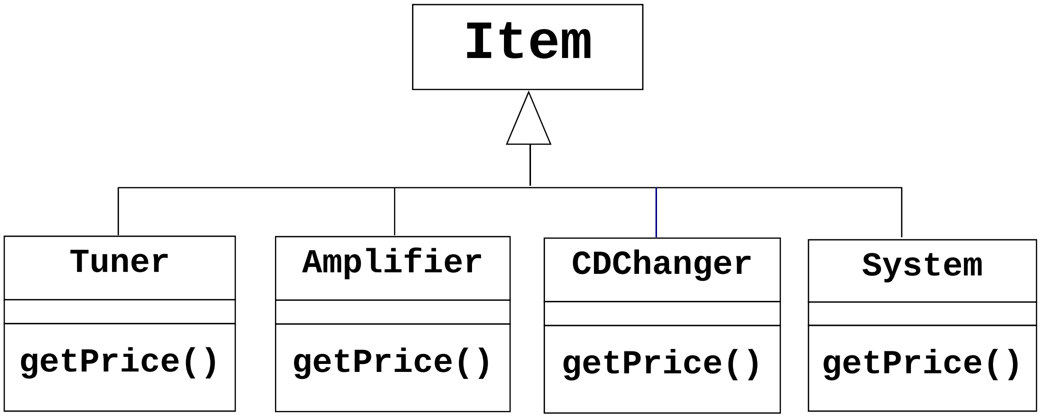
The Composite Pattern

- **MOTIVATION:**
  - Objects organized in a hierarchical manner
  - Some objects are *groups* of the other objects
  - Individuals and groups need to be treated uniformly
- **Example:**
  - A store sells stereo component items:

    Tuners, Amplifiers, CDChangers, etc.

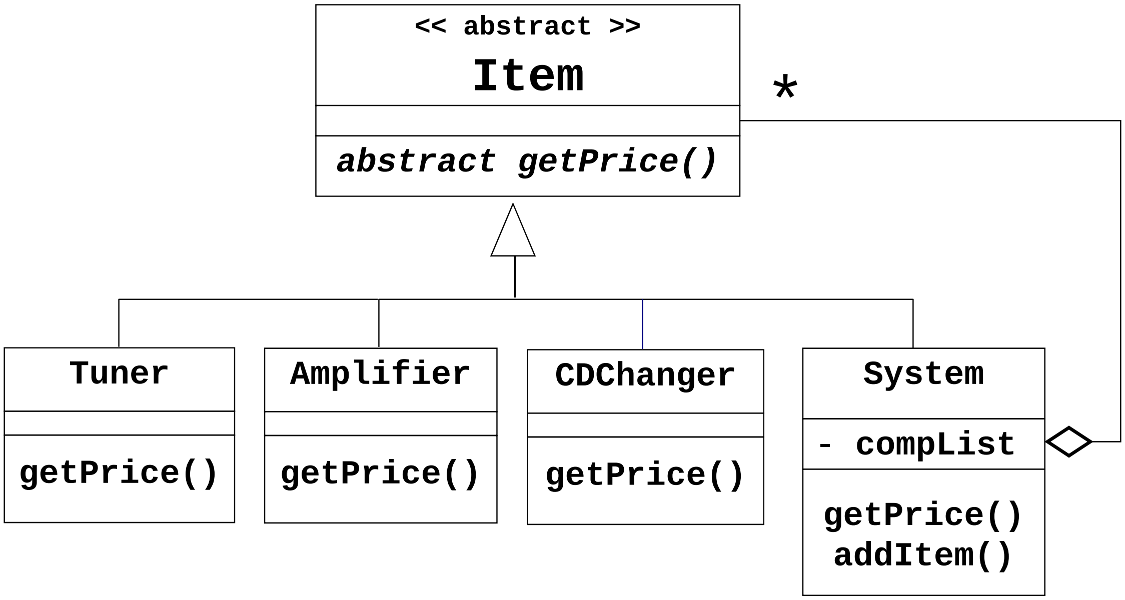
    - Each item has a **`getPrice()`** method
  - The store also sells complete stereo systems
    - Systems also have a **`getPrice()`** method which returns a discounted sum of the prices.

Possible Class Organization

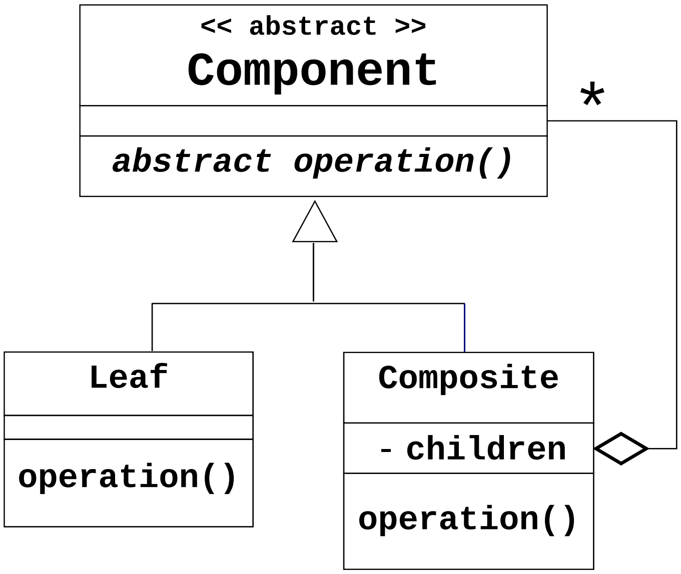


Problem ?

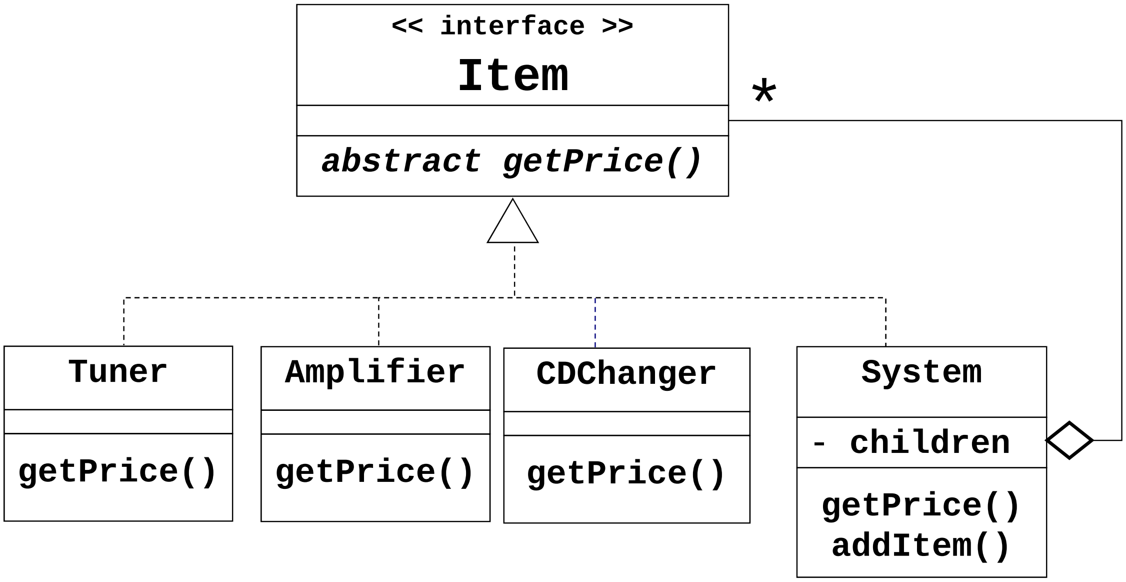
Solution



Composite Pattern Organization



Composite Specified With *Interfaces*



Other Examples Of Composites

- **Trees**
  - **Internal nodes (groups) and leaves**
- **Arithmetic expressions**
  - **`<exp> ::= <term> | <term> “+” <exp>`**
- **Graphical Objects**
  - **Rectangles, lines, circles**
  - **Frames**
    - **can contain other graphical objects**

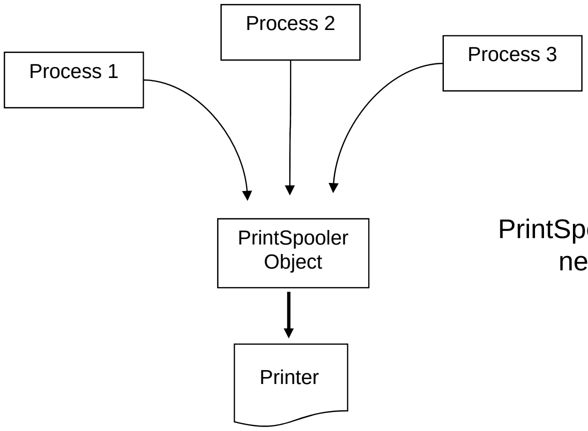
The Singleton Pattern

- **Motivation**
  - Insure a class never has more than one instance at a time
  - Provide public access to instance creation
  - Provide public access to current instance

- **Examples**
  - Print spooler
  - Audio player

PrintSpooler Example

Multiple processes should not access a single printer simultaneously



PrintSpooler mySpooler =
new PrintSpooler();

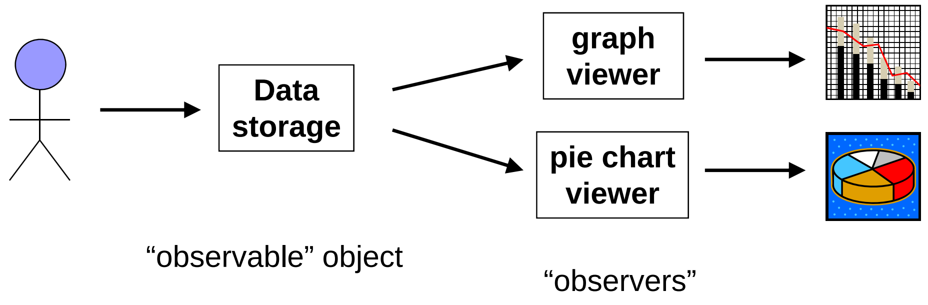
Singleton Implementation

```
public class PrintSpooler {
    // maintain a single global reference to the spooler
    private static PrintSpooler theSpooler;

    // insure that no one can construct a spooler directly
    private PrintSpooler()   { }

    // provide access to the spooler, creating it if necessary
    public static PrintSpooler getSpooler() {
      if (theSpooler == null)
            theSpooler =  new PrintSpooler();
      return theSpooler;
    }

    // accept a Document for printing
    public void addToPrintQueue (Document doc) {
      //code here to add the Document to a private queue ...
    }

    //private methods here to dequeue and print documents ...
}
```

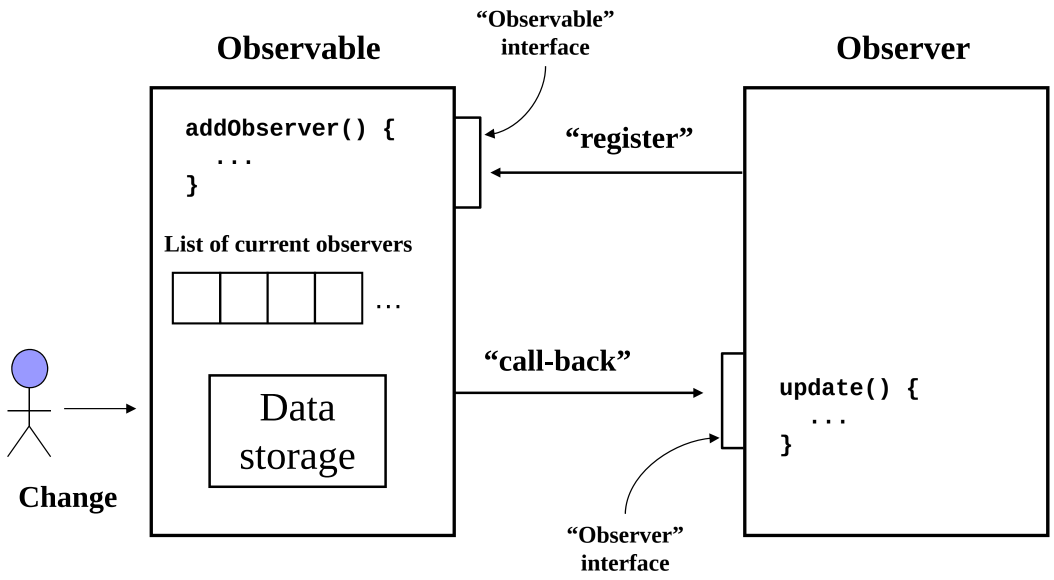
The Observer Pattern

Motivation

- An object stores data that changes regularly
- Various clients use the data in different ways
- Clients need to know when the data changes
- Data storage shouldn't have to keep track of clients



The Observer Pattern (cont.)



```
addObserver() {
   ...
}
```

```
update() {
   ...
}
```

Responsibilities

- Observables must
  - Provide a way for observers to “register”
  - Keep track of who is “observing” them
  - *Notify observers* when something changes
- Observers must
  - Tell observable it wants to be an observer *(“register”)*
  - Provide a method for the *callback*
  - Decide what to do when notified an observable has changed

Implementing Observer/Observable

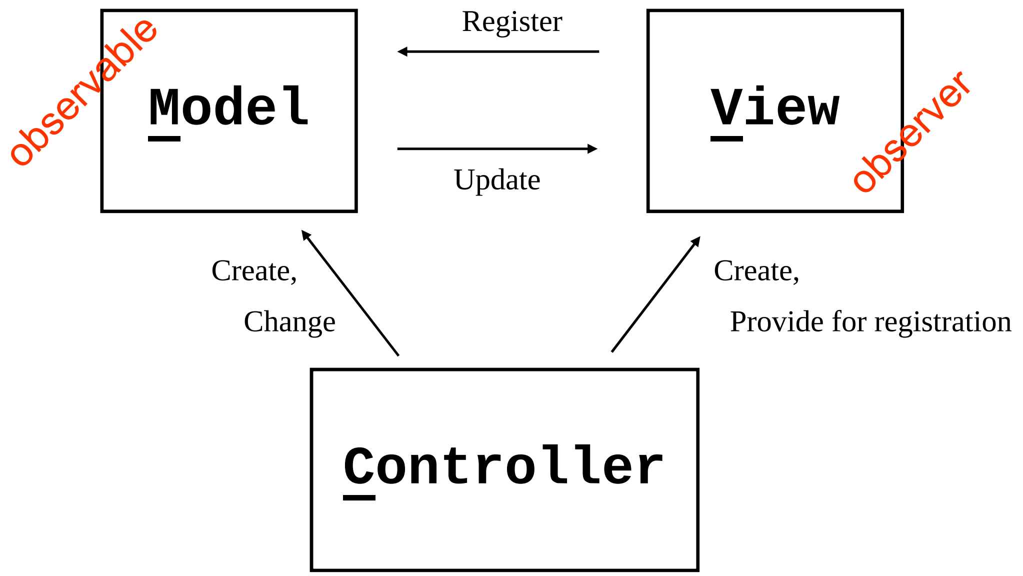
```
public interface Observer {
   public void update (Observable o, Object arg);
}
```

```
public interface Observable {
   public void addObserver (Observer obs);
   public void notifyObservers();
}
```

OR…

```
public class Observable extends Object {
   public void addObserver (Observer obs) {...}
   public void notifyObservers() {...}
   protected void setChanged() {...}
   ...
}
```

MVC Architecture



Model

View

Controller

Register

Update

Create,

Change

Create,

Provide for registration

observable

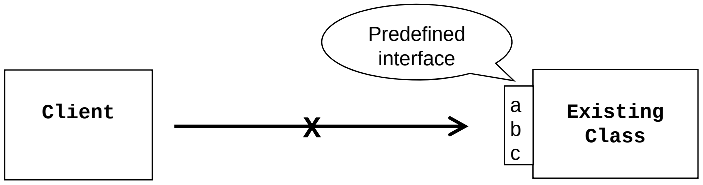
observer

```
public class Controller {
  private Model model;
  private View v1;
  private View v2;
  public Controller () {
    model = new Model();    // create “Observable” model
    v1 = new View(model);   // create an “Observer” that registers itself
    v2 = new View();        // create another “Observer”
    model.addObserver(v2); // register the observer
  }
  // methods here to invoke changes in the model
}

public class Model implements Observable {
  // declarations here for all model data...
  // methods here to manipulate model data, handle
  // observer registration, invoke observer callbacks, etc.
}

public class View implements Observer {
  public View(Observable myModel) {  // this constructor also
     myModel.addObserver(this);      // registers itself as an Observer
  }
  public View ()
  { }  // this constructor assumes 3rd-party Observer registration
  public update (Observable o, Object arg) {
    // code here to output a view based on the data in the Observable
  }
}
```

The Adapter Pattern

Motivation:

- **A client wants to use an *existing* class.**
- **The client wants to use a certain interface.**
- **The existing class provides a *different* interface.**



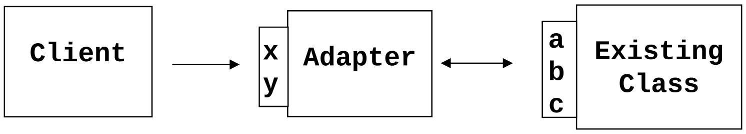
The Adapter Pattern (cont.)

Example contexts :

- Need to add an *Icon* to a *Container* – containers only hold *Components* (icons don’t implement component interface)
- A class has method parameters with “U.S. Standard” units; a client wants to use the methods with Metric units.
- A client wants a simple “*WindowClosingListener*”, but the WindowListener interface requires six different window operation methods to be implemented.
- An application supports “plugins”, but an existing class does not provide the proper “plugin interface”.

The Adapter Pattern (cont.)

- Define an ***adapter*** class implementing the desired ("target") interface.
- Client interacts with the adapter.
- Adapter translates target calls into equivalent calls to the existing class ("adaptee")



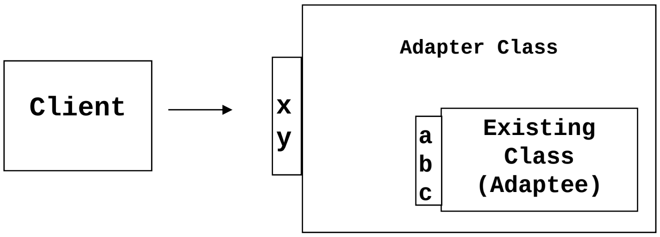
The Adapter Pattern (cont.)

Implementation alternatives:

1. Encapsulation of adaptee
   - "Object adapter"
2. Subclassing (inheritance)
   - "Class adapter"

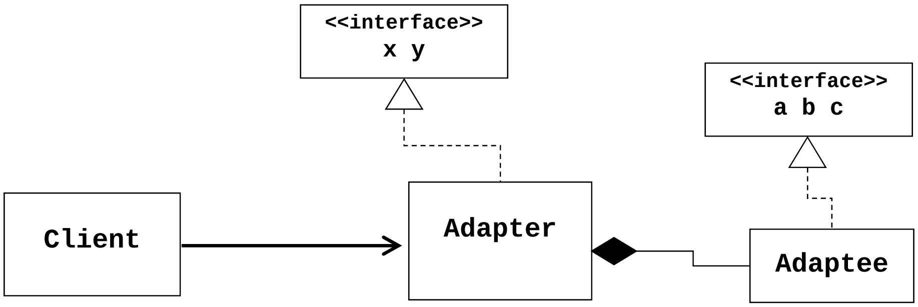
The Adapter Pattern (cont.)

- Encapsulation approach
  - Adapter owns an instance of Adaptee



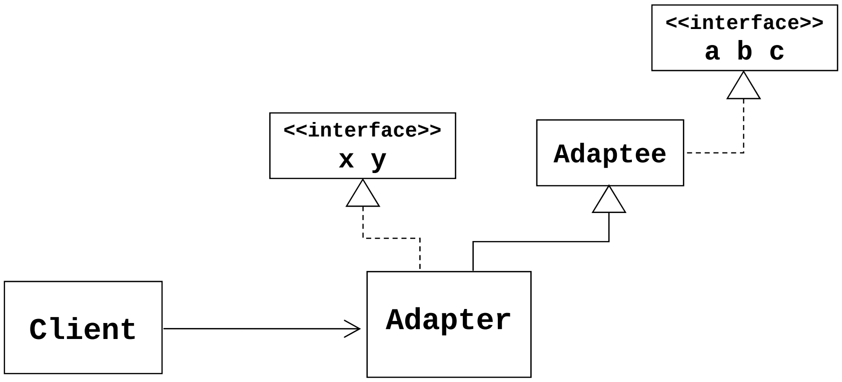
The Adapter Pattern (cont.)

- UML for encapsulation approach



The Adapter Pattern (cont.)

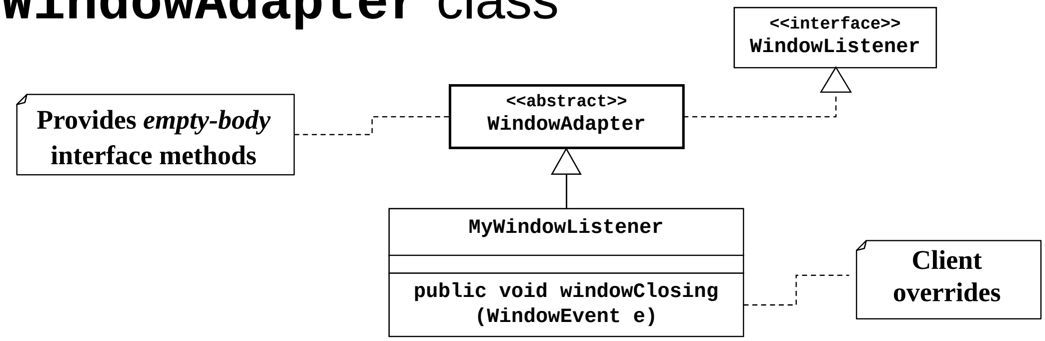
- Subclassing approach
  - Adapter *EXTENDS* Adaptee
  - UML:



- Advantages of encapsulation approach
  - Adapter class can extend another class
  - Adapter class can adapt multiple adaptees
  - Adaptee methods are invisible to client
- Advantages of subclassing approach
  - Adapter class can inherit methods from adaptee
  - Adaptee methods are visible to client

Java Adapters

- **`WindowAdapter`** class



- Similar classes for most event listeners:
  - **`MouseAdapter`**
  - **`KeyAdapter`**
  - **`ComponentAdapter`**
  - **`ContainerAdapter`**

Anonymous Adapter

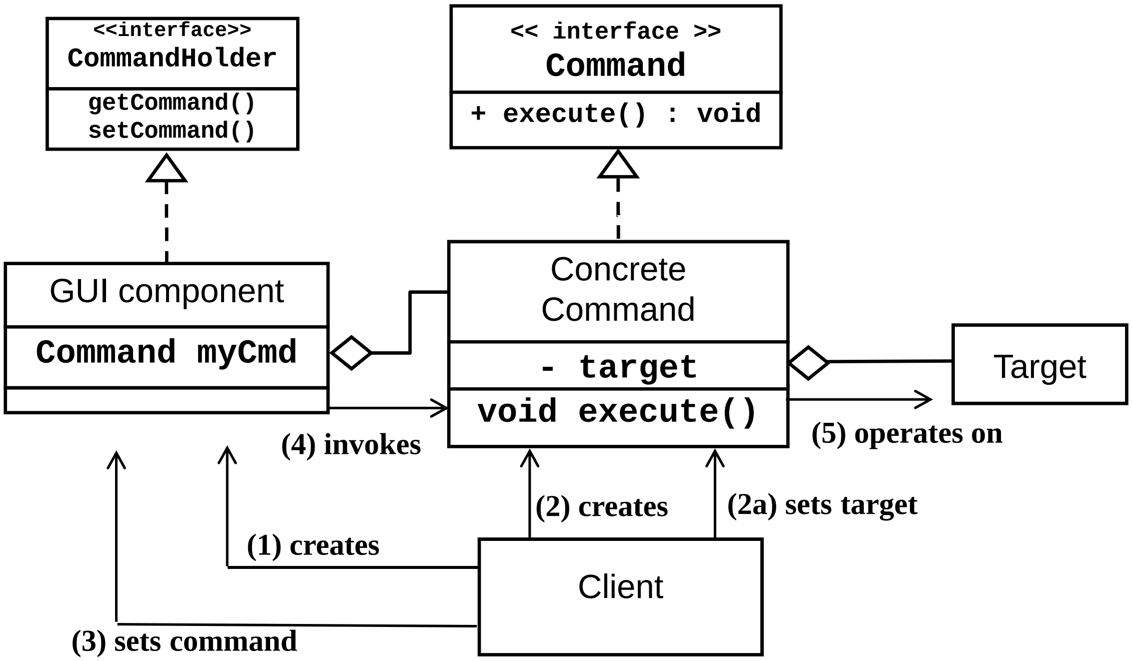
```
/** A JFrame which uses an anonymous class for Window Closing */
public class AnonymousAdapterFrame extends JFrame implements ActionListener {
  public AnonymousAdapterFrame () {
    // ...code here to initialize the frame as usual...
    // create a new button and add it to the frame
    JButton doneButton = new JButton ("Push Me To Quit");
    this.add(doneButton);
    // attach an action listener to the button:  point back to here
    doneButton.addActionListener(this);
    // attach an anonymous extended adapter as a window listener
    this.addWindowListener(
      new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          System.exit(0);
        }
      }
    );
    setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
    setVisible(true);
  }
  // button action handler -- terminates the program
  public void actionPerformed (ActionEvent e) {
    System.exit(0);
  }
}
```

The Command Pattern

Motivation

- Desire to separate code implementing a command from the object which invokes it
- Need for maintaining *state information* about the command
  - Enabled or disabled?
  - Invocation history (e.g. to support “undo”)
  - Other data – e.g. invocation *count*
- Need for a *collection* of commands that can be processed polymorphically

Command Pattern Organization



<<interface>>
CommandHolder
getCommand()
setCommand()

<< interface >>
Command
+ execute() : void

GUI component
Command myCmd

Concrete Command
- target
void execute()

Target

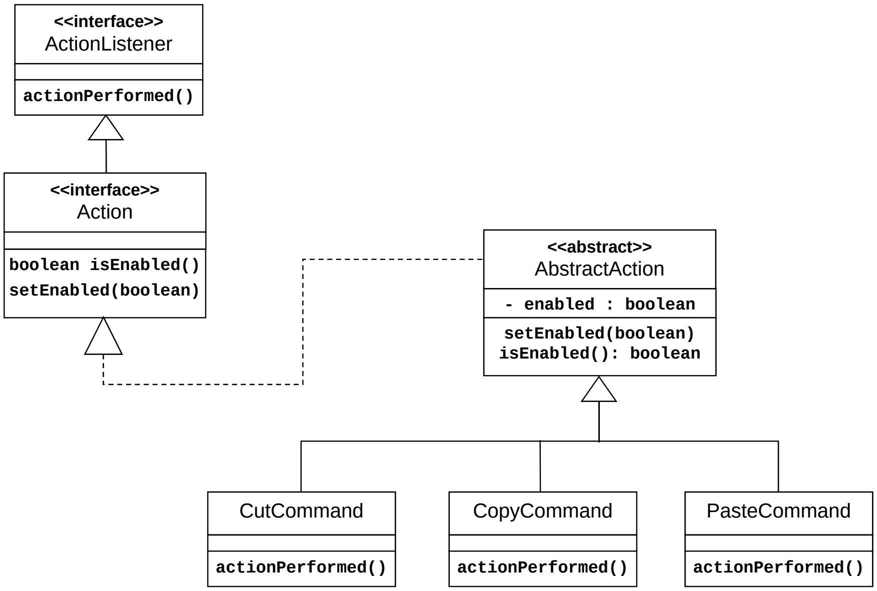
Client

(1) creates
(2) creates
(2a) sets target
(3) sets command
(4) invokes
(5) operates on

Java “Command” Analogue

- **`Action`** interface
  - Defines methods provided by Action (“command”) objects
    - **`isEnabled()`**
    - **`setEnabled()`**
    - **`actionPerformed()`** == “**`execute()`**”
    - …
- **`AbstractAction`** class
  - Implements “**`Action`**”
  - Provides *implementations* of *most* **`Action`** methods
    - except: **`actionPerformed()`**
  - Makes it simple to create “Actions” (commands):
    1. Extend **`AbstractAction`**
    2. Implement **`actionPerformed()`**

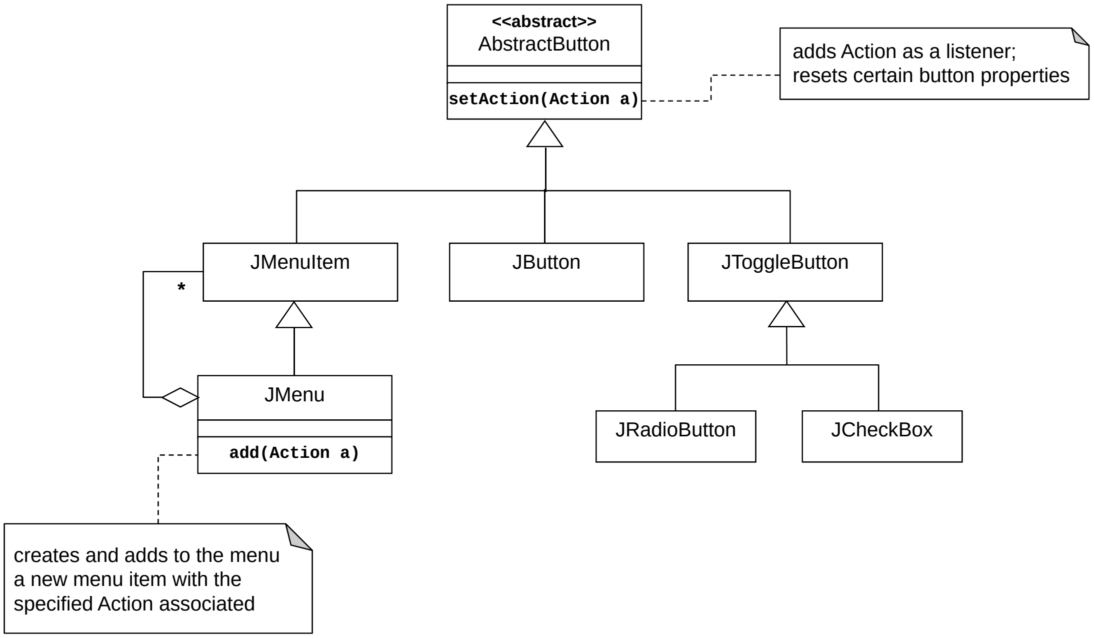
Java AbstractActions



Java “CommandHolder” Analogue

- **`AbstractButton` class**
  - Ancestor of components generating **`ActionEvents`**
    - **`JMenus`, `JMenuItems`, `JButtons ...`**
  - Supports holding “Actions” (commands):
    - do this with: **`setAction(Action a)`**
  - the commands (Actions) are *automatically added as listeners on the component*
  - GUI components automatically invoke the Actions

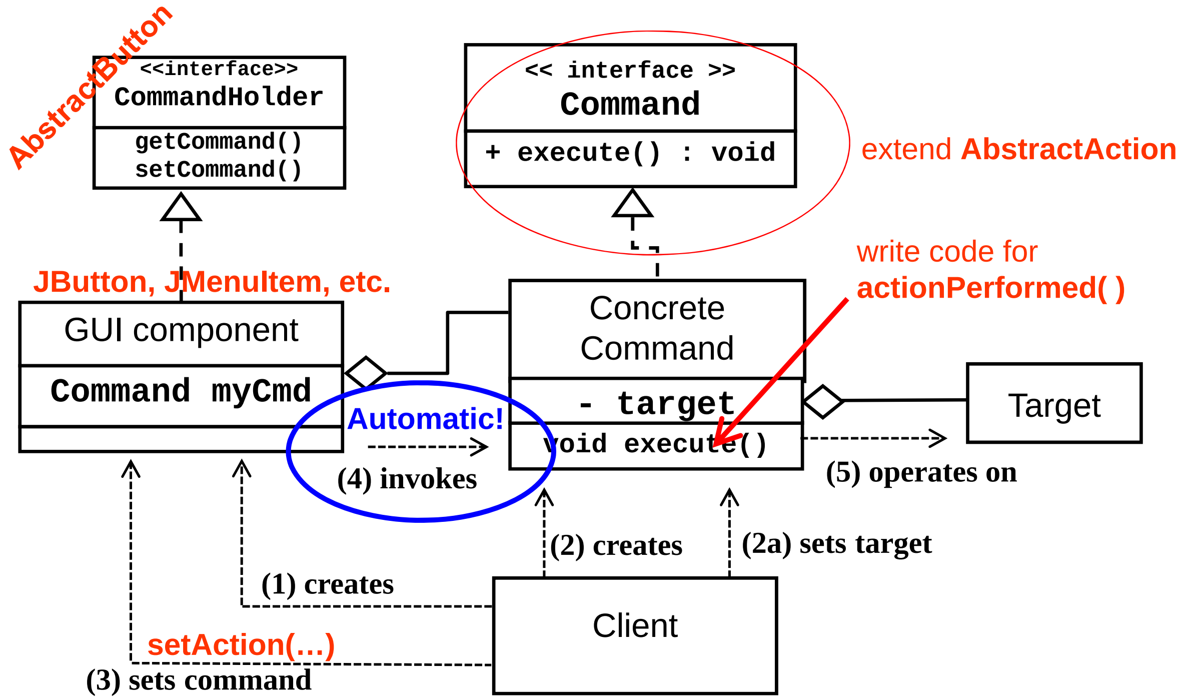
Java `AbstractButtons`



adds Action as a listener; resets certain button properties

creates and adds to the menu a new menu item with the specified Action associated

Command Pattern - Java



AbstractButton

<<interface>>
CommandHolder
getCommand()
setCommand()

<< interface >>
Command
+ execute() : void

extend **AbstractAction**

write code for
actionPerformed( )

JButton, JMenuItem, etc.

GUI component
Command myCmd

Concrete
Command
- target
void execute()

Target

Automatic!

(4) invokes

(5) operates on

(2) creates

(2a) sets target

(1) creates

Client

setAction(…)

(3) sets command

Using Java *Actions* : Summary

- **Define *Command* classes:**
    - Extend **`AbstractAction`**
    - Implement (write code for) **`actionPerformed()`**
- **Create a `JFrame`**
- **Add components to frame (`JButtons`, `JMenuItems`,…)**
- **Instantiate Command (“Action”) objects**
- **Add Command objects to components w/ `setAction()`**
    - **`JMenus`** allow automatic MenuItem creation using **`add(Action)`**

Using Java *Actions*: Example

```
/** This class defines a Command which performs "cut" operations.
 *  The command is implemented as an AbstractAction, allowing it
 *  to be added to any object supporting attachment of Actions.
 *  This example does not show how the “Target” of the command is specified.
 */
public class CutCommand extends AbstractAction {

  public CutCommand() {

    super("Cut");
  }

  //invoked to perform the 'cut' operation
  public void actionPerformed (ActionEvent e) {
    //...code here to perform the actual “cut” operation...
    System.out.println ("Cut action invoked from " +
                        e.getActionCommand() + " " +
                        e.getSource().getClass() );
  }
}
```

```
/** This class defines a Command which performs the “Quit” operation. It prompts
 *  for confirmation of the intent to quit, and then exits if the user confirms.
 */
public class QuitCommand extends AbstractAction {
  public QuitCommand() {
    super("Quit");
  }
  public void actionPerformed (ActionEvent e) {
    // display the source of the request
    System.out.println ("Quit requested from " + e.getActionCommand()
                  + " " + e.getSource().getClass() );
    // verify intent before exiting by displaying a Yes/No dialog
    int result = JOptionPane.showConfirmDialog
      (null,                                    // source of event
      "Are you sure you want to exit  ?",       // display message
      "Confirm Exit",                           // Title bar text
      JOptionPane.YES_NO_OPTION,                // button choices
      JOptionPane.QUESTION_MESSAGE);            // prompt icon

    if (result == JOptionPane.YES_OPTION) {
      System.exit(0);
    }
    return;    // we get here if "No" was chosen
  }
}
```

```
/** This class instantiates several AbstractAction command objects, creates several GUI
 *  components, and attaches the command objects to the GUI components.  The command
 *  objects then automatically get invoked when the GUI component is activated.
 */
public class MainFrame extends JFrame {
  public MainFrame () {
    // ...code here to initialize the frame
    // create a menu bar with "File" and "Edit" menus
    JMenuBar bar = new JMenuBar();
    setJMenuBar(bar);
    JMenu fileMenu = new JMenu("File");
    bar.add(fileMenu);
    JMenu editMenu = new JMenu("Edit");
    bar.add(editMenu);
    // create AbstractActions (commands) to attach to File menu items
    OpenCommand openCommand = new OpenCommand();
    SaveCommand saveCommand = new SaveCommand();
    QuitCommand quitCommand = new QuitCommand();
    // create menu items File->Open/Save/Quit
    JMenuItem fileOpen = new JMenuItem("Open");
    JMenuItem fileSave = new JMenuItem("Save");
    JMenuItem fileQuit = new JMenuItem ("Quit");
    // ...continued
```

```
    fileMenu.add(fileOpen);        //add menu items to File menu
    fileMenu.add(fileSave);
    fileMenu.add(fileQuit);
    // Add Commands to File menu items. Notice that no explicit "addListener"
    // is needed; they automatically become listeners when added to GUI components
    fileOpen.setAction(openCommand);
    fileSave.setAction(saveCommand);
    fileQuit.setAction(quitCommand);

    CutCommand cutCommand = new CutCommand();     // create AbstractActions (Commands)
    CopyCommand copyCommand = new CopyCommand(); // to be attached to menu items
    PasteCommand pasteCommand = new PasteCommand();
    editMenu.add(cutCommand);      // Automatically create Edit menu items
    editMenu.add(copyCommand);     // and implicitly add command listeners
    editMenu.add(pasteCommand);    //  (instead of using setAction())
    JButton cutButton = new JButton("Cut");   // Add some buttons to the frame
    this.add(cutButton);
    // Add the (same) Command objects to the buttons. Note again that no explicit
    // "addListener" for the buttons is needed;  The AbstractAction Command objects
    // become registered listeners when added to the buttons.
    cutButton.setAction(cutCommand);
    copyButton.setAction(copyCommand);
    pasteButton.setAction(pasteCommand);
    quitButton.setAction(quitCommand);
  }
}
```

Run

The Decorator Pattern

MOTIVATION:

- You want to enhance the behavior of a class ***at run time***
- You want to otherwise be able to treat normal and decorated instances interchangeably

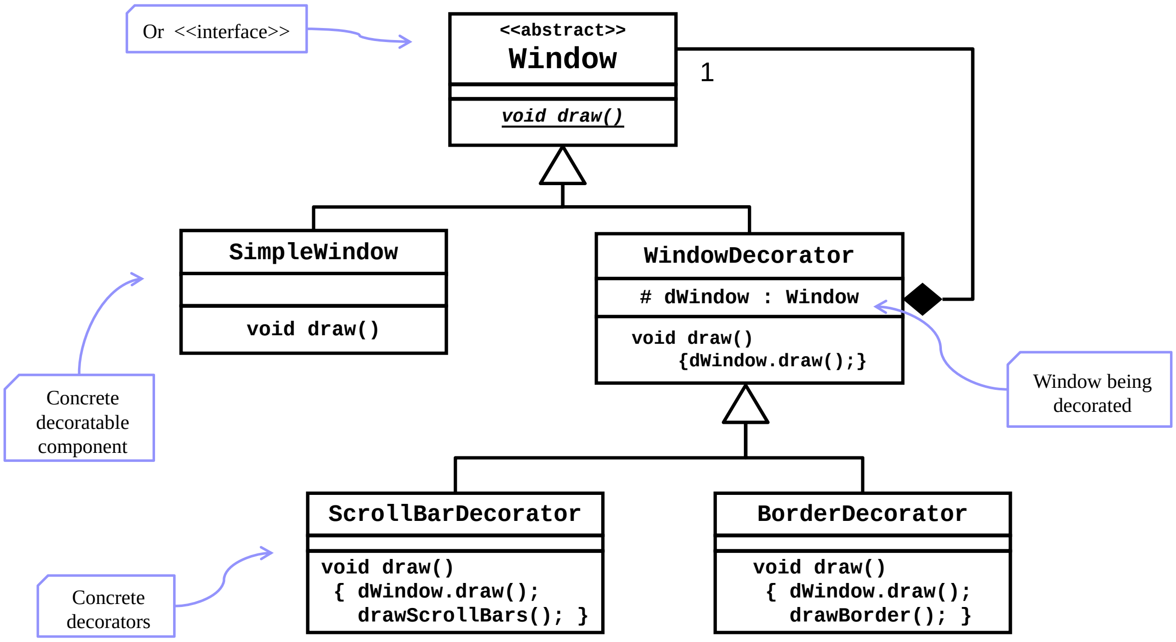
Examples:

- Bricks on a house (or in a game) are normally red, but sometimes can be adorned with additional markings
- A FileReader is wrapped inside a BufferedReader
- A TextArea is decorated with ScrollBars, or a Border, or both
- A file icon is decorated with status markings such as “in sync with repository”, “has warnings”, “has errors”, “is read-only”

Decorators vs. Subclassing

- Assume an existing class “Window”, and we want to support variations – e.g.
  - Windows with scrollbars
  - Windows with borders
  - Windows with *both*
- We want to be able to switch dynamically (e.g. add or remove various decorations)
- Defining derived subclasses is complex
  - Sub-classes **`WindowWithBorder`**, **`WindowWithScrollBars`**, **`WindowWithBorderAndScrollBars`**, … ??
  - Problem gets worse every time a new “decoration feature” is needed…

Window Decorator Pattern



Or <<interface>>

<<abstract>>
Window

void draw()

1

SimpleWindow

void draw()

WindowDecorator

# dWindow : Window

void draw()
{dWindow.draw();}

ScrollBarDecorator

void draw()
{ dWindow.draw();
drawScrollBars(); }

BorderDecorator

void draw()
{ dWindow.draw();
drawBorder(); }

Concrete decoratable component

Window being decorated

Concrete decorators

Using Decorators

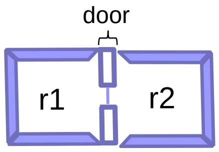
```
//create a window decorated with just scrollbars
Window scrollbarWindow = new ScrollBarDecorator( new SimpleWindow() ) ;
scrollbarWindow.draw();

//create a window decorated with just a border
Window borderedWindow = new BorderDecorator( new SimpleWindow() ) ;
borderedWindow.draw();

//create a window decorated with scrollbars and border
Window decoratedWindow = new BorderDecorator(
                              new ScrollBarDecorator( newSimpleWindow() ) );
decoratedWindow.draw();

//create a buffered reader for a text file
BufferedReader in = new BufferedReader( new FileReader( new File("file.txt") ) );
//read and echo data from the buffered file
String inString = in.readLine();
while (! (inString == null)) {
    System.out.println (inString);
    inString = in.readLine();
}
```

The Factory Method Pattern

- **Motivation**
  - **Sometimes a class can’t anticipate the class of objects it must create**
  - **It is sometimes better to delegate specification of object types to subclasses**
  - **It is frequently desirable to avoid binding application-specific classes into a set of code**

Example: Maze Game

```
public class MazeGame {

    // This method creates a maze for the game, using a hard-coded structure for the
    // maze (specifically, it constructs a maze with two rooms connected by a door).
    public Maze createMaze () {
        Maze theMaze = new Maze() ;   //construct an (empty) maze
        Room r1 = new Room(1) ;       //construct components for the maze
        Room r2 = new Room(2) ;
        Door theDoor = new Door(r1, r2);

        r1.setSide(NORTH, new Wall()); //set wall properties for the rooms
        r1.setSide(EAST,  theDoor);
        r1.setSide(SOUTH, new Wall());
        r1.setSide(WEST,  new Wall());
        r2.setSide(NORTH, new Wall());
        r2.setSide(EAST,  new Wall());
        r2.setSide(SOUTH, new Wall());
        r2.setSide(WEST,  theDoor);

        theMaze.addRoom(r1); //add the rooms to the maze
        theMaze.addRoom(r2);

        return theMaze ;
    }
    //other MazeGame methods here (e.g. a main program which calls createMaze())...
}
```



Based on an example in "*Design Patterns: Elements of Reusable Object-Oriented Software*" by Gamma et. al. (the so-called "Gang of Four" book).

Problems with `createMaze()`

- Inflexibility; lack of “reusability”
- Reason: it “hardcodes” the maze types
  - Suppose we want to create a maze with (e.g.)
    - Magic Doors
    - Enchanted Rooms
  - Possible solutions:
    - Subclass MazeGame and override **`createMaze()`** (i.e., create a whole new version with new types)
    - Hack **`createMaze()`** apart, changing pieces as needed

`createMaze()` Factory Methods

```
public class MazeGame {

    //factory methods - each returns a MazeComponent of a given type
    public Maze makeMaze()        { return new Maze() ; }
    public Room makeRoom(int id)  { return new Room(id) ; }
    public Wall makeWall()        { return new Wall() ; }
    public Door makeDoor(Room r1, Room r2) { return new Door(r1,r2) ; }

    // Create a maze for the game using factory methods
    public Maze createMaze () {
        Maze theMaze = makeMaze() ;
        Room r1 = makeRoom(1) ;
        Room r2 = makeRoom(2) ;
        Door theDoor = makeDoor(r1, r2);
        r1.setSide(NORTH, makeWall());
        r1.setSide(EAST,  theDoor);
        r1.setSide(SOUTH, makeWall());
        r1.setSide(WEST,  makeWall());
        r2.setSide(NORTH, makeWall());
        r2.setSide(EAST,  makeWall());
        r2.setSide(SOUTH, makeWall());
        r2.setSide(WEST,  theDoor);
        theMaze.addRoom(r1);
        theMaze.addRoom(r2);
        return theMaze ;
    }
    ...
}
```

Overriding Factory Methods

```
//This class shows how to implement a maze made of different types of rooms.  Note
// in particular that we can call exactly the same (inherited) createMaze() method
// to obtain a new “EnchantedMaze”.
public class EnchantedMazeGame extends MazeGame {

    //override MakeRoom to produce "EnchantedRooms"
    @Override
    public Room makeRoom(int id) {

        //create the spell necessary to enter the enchanted room
        Spell spell = makeSpell() ;

        //construct and return an EnchantedRoom requiring a spell to be entered
        return new EnchantedRoom(id, spell);
    }

    //override MakeDoor to produce a door requiring a spell
    @Override
    public Door makeDoor(Room r1, Room r2) {

        //construct and return a Door requiring a spell to be entered
        return new DoorNeedingSpell(r1, r2);
    }

    //new factory method for making spells
    public Spell makeSpell() { return new Spell() ;}
    ...
}
```

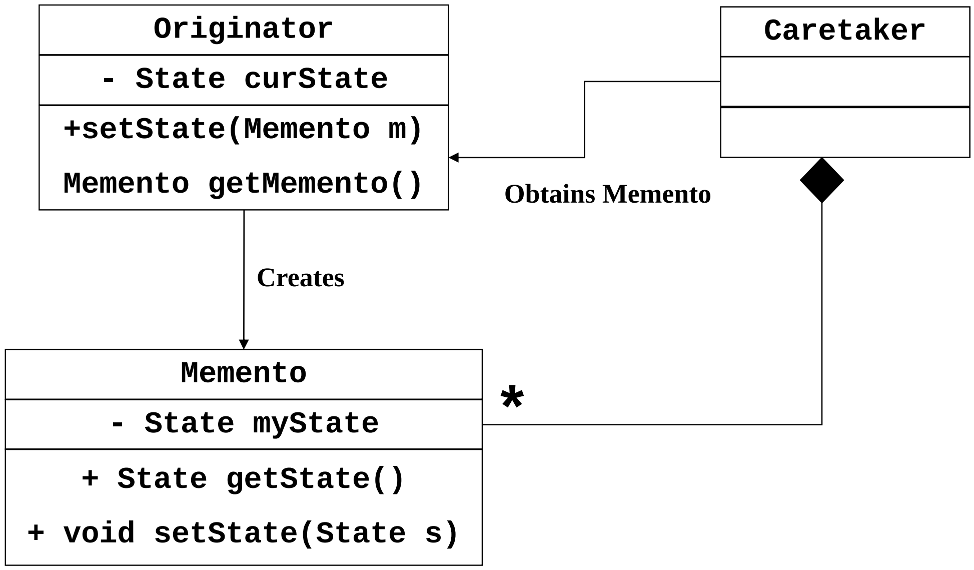
The Memento Pattern

- Motivation: support “undo” and/or “restart”
  - restore some object(s) to a previous state
- Requirements:
  - Each object keeps track of its state changes, or
  - Some external object accesses and tracks states
- Problems:
  - Self-tracking can cause large, cumbersome objects
  - External tracking requires *access to internal state*
    - Violates encapsulation!

The Memento Pattern (cont.)

- Intent: externalize an object’s internal state
  - Maintain encapsulation
  - Allow returning an object to a prior state
- Participants:
  - “Originator”
    - The object whose state needs to be captured
  - “Memento”
    - An object holding the Originator’s state
  - “Caretaker”
    - An object that acquires and manages mementos

Memento Pattern Organization



Benefits / Drawbacks

- Preserves encapsulation
- Simplifies Originator
  - Doesn't have to keep track of states
- Can be expensive
  - Potentially lots of state information
    - Mitigation: store only *incremental state changes* (or *limit the number of "undo's" allowed)*
- Can impose large load on Caretaker

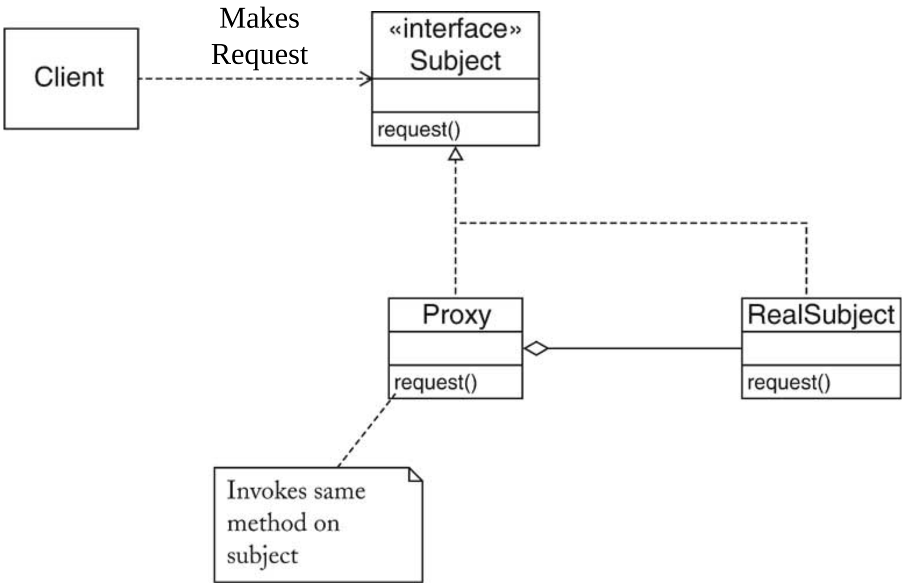
The `Proxy` Pattern

- Motivation
  - Undesirable target object manipulation
    - Access required, but not to all operations
  - Expensive target object manipulation
    - Lengthy image load time
    - Significant object creation time
    - Large object size
  - Inaccessible target object
    - Resides in a different address space
      - E.g. another JVM or a machine on a network

Proxy Types

- Protection Proxy – controls access
- Virtual Proxy – acts as a stand-in
- Remote Proxy – local stand-in for object in another address space

Proxy Pattern Organization



Proxy Example

```
interface IGameWorld {
     Iterator getIterator();
     void addGameObject(GameObject o);
     boolean removeGameObject (GameObject o);
}


/**A proxy which prohibits removal of GameWorldObjects from the GameWorld*/
public class GameWorldProxy implements IObservable, IGameWorld {
     private GameWorld realGameWorld ;
     public GameWorldProxy (GameWorld gw)
       { realGameWorld = gw; }

     public Iterator getIterator ()
       { return realGameWorld.getIterator(); }

     public void addGameObject(GameObject o)
       {  realGameWorld.addGameObject(o) ;  }

     public boolean removeGameObject (GameObject o)
       {  return false ; }
}
```

Proxy Example (cont.)

```
/** This class defines a Game containing a GameWorld with a ScoreView Observer. */
public class Game {
    public Game() {
        GameWorld gw = new GameWorld();   //construct a GameWorld
        ScoreView sv = new ScoreView();   //construct a ScoreView
        gw.addObserver(sv);               //register ScoreView as a GameWorld Observer
    }
}
----------------------------------------------------------------------------------
/** This class defines a GameWorld which is an Observable and maintains a list of
 *  Observers; when the GameWorld needs to notify its Observers of changes it does so
 *  by passing a GameWorldProxy to the Observers. */
public class GameWorld implements IObservable, IGameWorld {
    private Vector<GameObject> myGameObjectList = new Vector<GameObject>();
    private Vector<IObserver> myObserverList = new Vector<IObserver>();
    public Iterator<GameObject> getIterator() { ... }
    public void addGameObject(GameObject o) { ... }
    public boolean removeGameObject(GameObject o) {
        //code here to remove the specified GameObject from the GameWorld...
    }
    public void addObserver(IObserver o) { myObserverList.add(o); }

    //Pass a GameWorldProxy to Observers, thus prohibiting Observer removal of GameObjects
    public void notifyObservers() {
        GameWorldProxy proxy = new GameWorldProxy(this);
        for (IObserver o : myObserverList) {
            o.update((IObservable)proxy, null);
        }
    }
}
```

The Strategy Pattern

Motivation

- A variety of algorithms exists to perform a particular operation
- The client needs to be able to select/change the choice of algorithm *at run-time.*

The Strategy Pattern (cont.)

Examples where different *strategies* might be used:

- Save a file in different formats (plain text, PDF, PostScript…)
- Compress a file using different compression algorithms
- Sort data using different sorting algorithms
- Capture video data using different encoding algorithms
- Plot the same data in different forms (bar graph, table, …)
- Have a game's non-player character (NPC) change its AI
- Arrange components in an on-screen window using different layout algorithms

Example: NPC AI Algorithms

Typical client code sequence:

```
void attack() {
         switch (characterType) {
         case WARRIOR:  fight();                break;
         case HUNTER:   fireWeapon();           break;
         case PRIEST:   castDisablingSpell();  break;
         case SHAMAN:   castMagicSpell();      break;
         }
}
```

Problem with this approach?

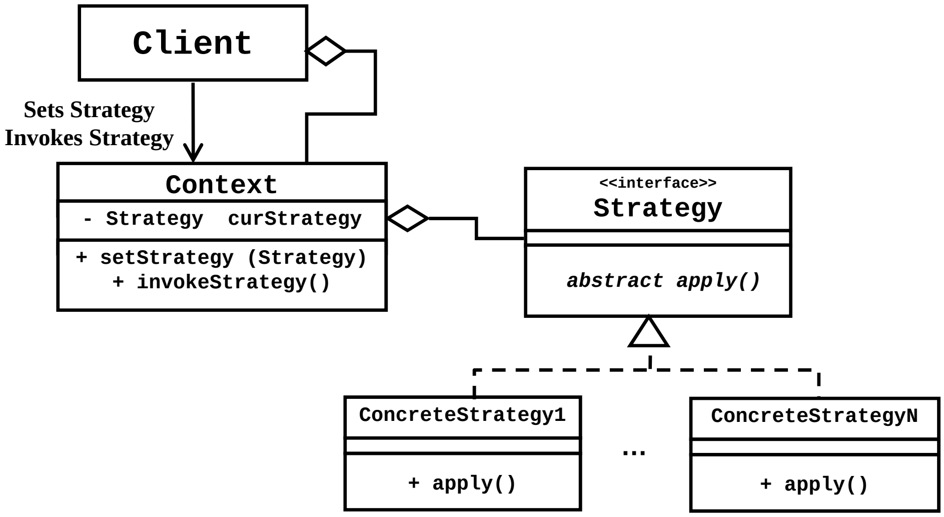
Changing or adding a plan requires changing the client!

Solution Approach

- Encapsulate the abstraction "*strategy*"
  - Capture the notions "*set strategy*" and "*apply strategy*"
- Provide various objects that know how to "apply strategy"
  - Each in a different way, but with a uniform interface
- Client maintains a "current strategy" object
- Provide a mechanism for the client code to *change* the current strategy object
  - Selection could be user-controlled, or based on client logic

Strategy Pattern Organization

- Using Interfaces



Client

Sets Strategy
Invokes Strategy

Context

- Strategy curStrategy

+ setStrategy (Strategy)
+ invokeStrategy()

<<interface>>
Strategy

abstract apply()

ConcreteStrategy1

+ apply()

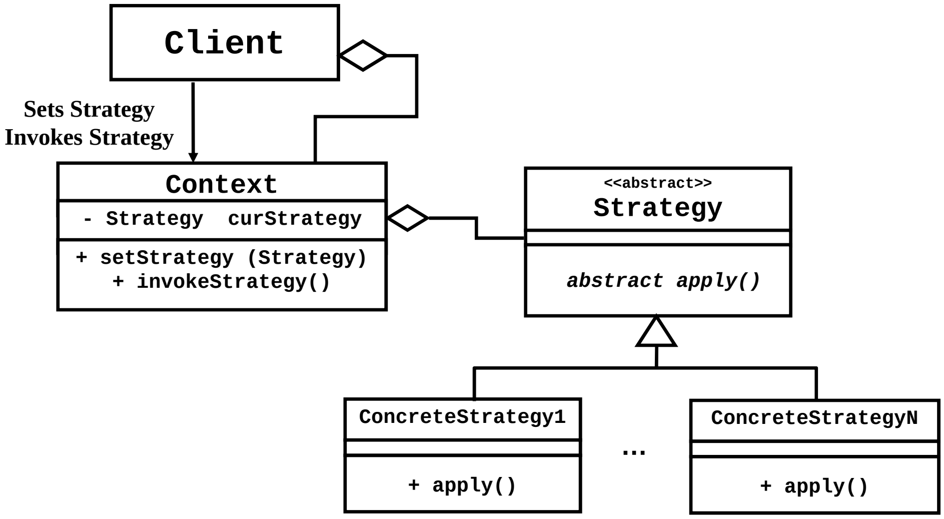
...

ConcreteStrategyN

+ apply()

Strategy Pattern Organization (cont.)

- Using subclassing



Client

Sets Strategy
Invokes Strategy

Context

- Strategy curStrategy

+ setStrategy (Strategy)
+ invokeStrategy()

<<abstract>>
Strategy

abstract apply()

ConcreteStrategy1

+ apply()

...

ConcreteStrategyN

+ apply()

Example: NPC’s in a Game

```
public interface Strategy {
    public void apply();
}

public class FightStrategy implements Strategy {
    public void apply() {
        //code here to do "fighting"
    }
}

public class FireWeaponStrategy implements Strategy {
    private Hunter hunter;
    public FireWeaponStrategy(Hunter h) {
        this.hunter = h;   //record the hunter to which this strategy applies
    }
    public void apply() {
        //tell the hunter to fire a burst of 10 shots
        for (int i=0; i<10; i++) {
            hunter.fireWeapon();
        }
    }
}
public class CastMagicSpellStrategy implements Strategy {
    public void apply() {
        //code here to cast a magic spell
    }
}
```

NPC’s in a Game (cont.)

“Contexts” :

```
public abstract class Character {
    protected Strategy curStrategy;
    abstract public void setStrategy(Strategy s);
    abstract public void invokeStrategy();
}
```

```
public class Warrior extends Character {
    public void setStrategy(Strategy s) {
        curStrategy = s;
    }
    public void invokeStrategy() {
        curStrategy.apply();
    }
}
```

```
public class Shaman extends Character {
    public void setStrategy(Strategy s) {
        curStrategy = s;
    }
    public void invokeStrategy() {
        curStrategy.apply();
    }
}
```

```
public class Hunter extends Character {
    private int bulletCount ;
    public void setStrategy(Strategy s) {
        curStrategy = s ;
    }
    public void invokeStrategy() {
        curStrategy.apply();
    }
    public boolean isOutOfAmmo() {
        if (bulletCount <= 0) return true;
        else return false;
    }
    public void fireWeapon() {
        bulletCount -- ;
    }
}
```

Assigning / Changing Strategies

```
/** This Game class demonstrates the use of the Strategy Design Pattern
 *  by assigning attack response strategies to each of several game characters.
 */
public class Game {
    //the list of non-player characters in the game
    ArrayList<Character> npcList = new ArrayList<Character>();
    public Game() {    //construct some characters, assigning each a starting strategy
        Warrior w1 = new Warrior();
        w1.setStrategy(new FightStrategy());
        npcList.add(w1);
        Hunter h1 = new Hunter();
        h1.setStrategy(new FireWeaponStrategy(h1));
        npcList.add(h1);
        Shaman s1 = new Shaman();
        s1.setStrategy(new CastSpellStrategy());
        npcList.add(s1);
    }
    public void attack() {    //force each character to execute its attack response
        for (Character c : npcList) {
            c.invokeStrategy();
        }
    }
    public void updateCharacters() {  //update any strategies that need changing
        for (Character c : npcList) {
            if(c instanceof Hunter) {
                if ( ((Hunter)c).isOutOfAmmo() ) {
                    //change the character's strategy
                    c.setStrategy(new FightStrategy());
                }
            }
        }
    }
}
```

Java Layout Managers

- Strategy Interface:

  `interface LayoutManager`

- “Context”:

  `class Container`

- Context methods:

  ```
  public void setLayout (LayoutManager mgr)
  public void doLayout()
  ```

- Concrete strategies:

  ```
  class FlowLayout()
  class BorderLayout()
  class GridLayout()
  ...
  ```

- “Apply” method:

  `void layoutContainer(Container parent)`

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

VIII – GUI Basics

- **Displays and Color**
- **Abstract Windowing Toolkit (AWT)**
- **Java Foundation Classes (JFC)**
- **Swing Components**
    - **Frames**
    - **Buttons**
    - **Labels**
    - **Check Boxes**
    - **Panels (Jpanels), etc.**
- **Layout Managers**
- **Building Menus**
- **Programmable Look-And-Feel (PLAF)**

CSc 133 Lecture Notes

VIII - GUI Basics

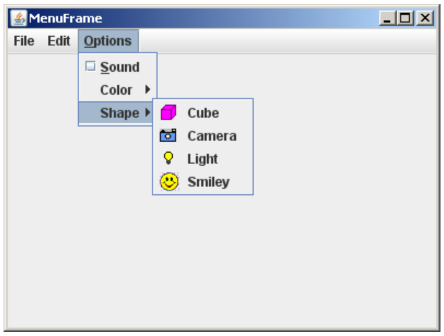
Computer Science Dept.
CSUS

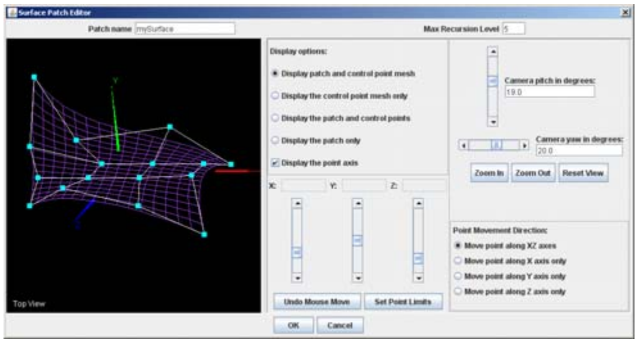
Overview

- Displays and Color
- The AWT and Swing Frameworks
- Swing Components
  - Frames, Buttons, Labels, Checkboxes, Dropdown Lists…
- Layout Managers
  - “Strategy” Design Pattern
- Panels
- Menus

Modern Program Characteristics

- Graphical User Interfaces (“GUIs”)
- “Event-driven” interaction





Patch name mySurface

Max Recursion Level 5

Display options:

● Display patch and control point mesh

○ Display the control point mesh only

○ Display the patch and control points

○ Display the patch only

☑ Display the point axis

X: Y: Z:

Camera pitch in degrees: 19.0

Camera yaw in degrees: 20.0

Point Movement Direction:

● Move point along XZ axes

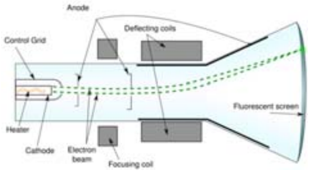
○ Move point along X axis only

○ Move point along Y axis only

○ Move point along Z axis only

Common Display Types

- CRT (Cathode Ray Tube)



- LCD (Liquid Crystal Display)

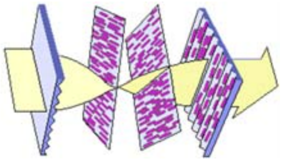
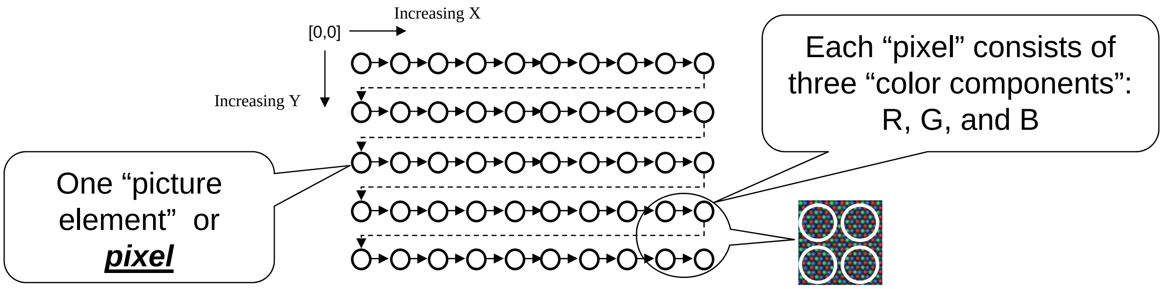


Image credits: Wikimedia Commons (http://commons.wikimedia.org); http://www.pctechguide.com/07panels.htm

Raster vs. Random Scan Devices

- Random scan: arbitrary movement
  - Oscilloscopes, pen-plotters, searchlights, laser light shows
- Raster scan: fixed (“raster”) pattern
  - CRTs, LCDs, Plasma panels, printers



Each “pixel” consists of three “color components”: R, G, and B

One “picture element” or ***pixel***

RGB Additive Color Model

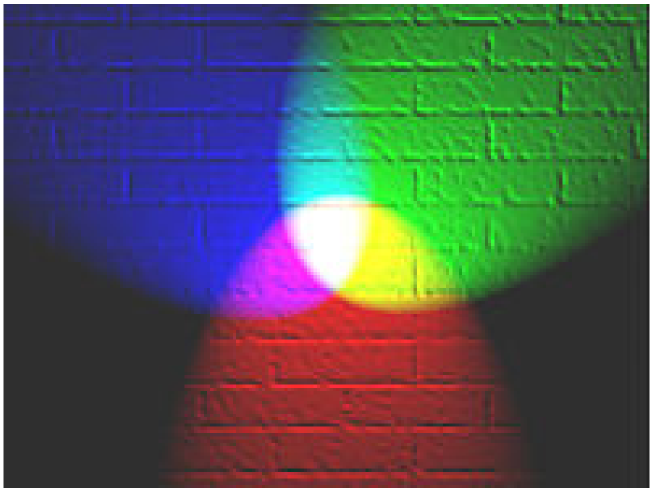


Image credit: http://en.wikipedia.org/wiki/RGB_color_model

The RGB Color Cube

- Each axis represents one of (Red, Green, Blue)
- Distance along axis = intensity (0 to max)
- Locations within cube = different colors
  - Values of equal RGB intensity are grey

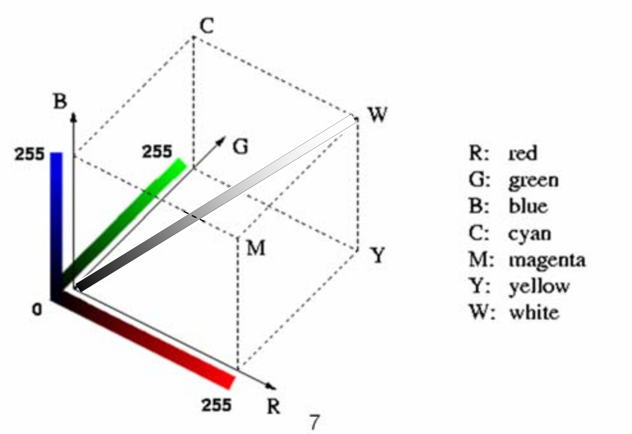
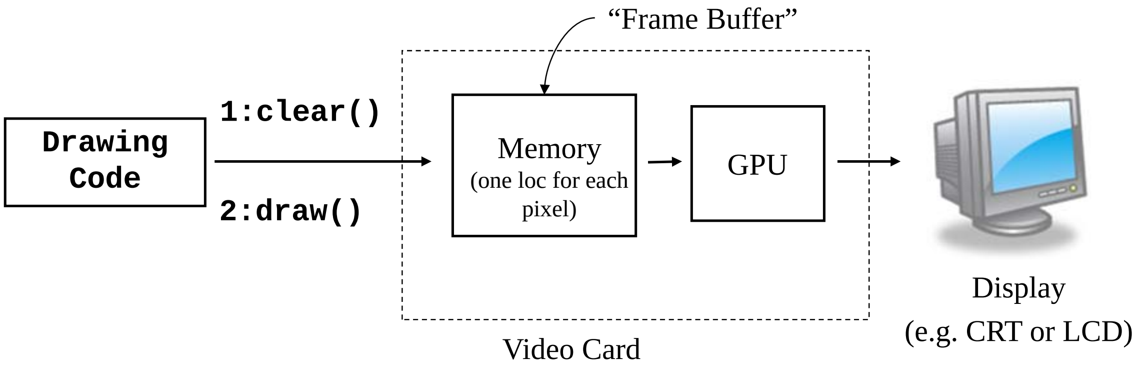


Image credit: http://gimp-savvy.com

Frame Buffers

- Program output is written to the video memory "*frame buffer*", in two steps (*clear*, then *draw*)
- Graphics Processing Unit copies memory contents (image) to display



"Frame Buffer"

Drawing Code

1:clear()

2:draw()

Memory (one loc for each pixel)

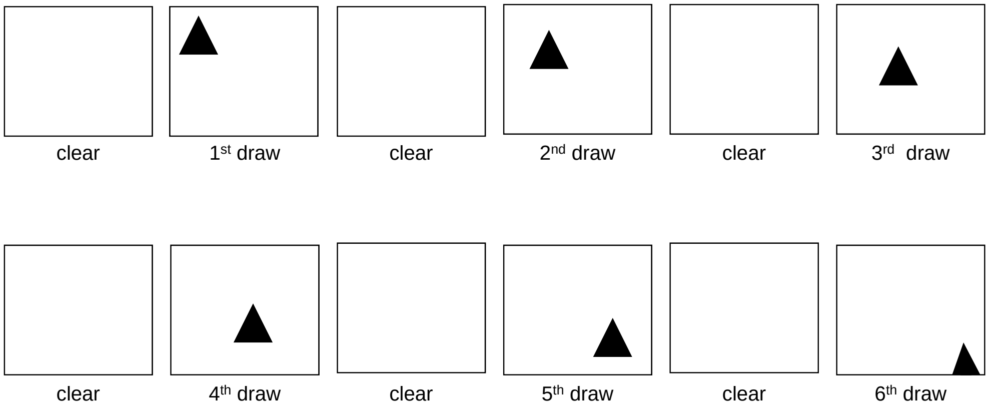
GPU

Video Card

Display (e.g. CRT or LCD)

Flicker

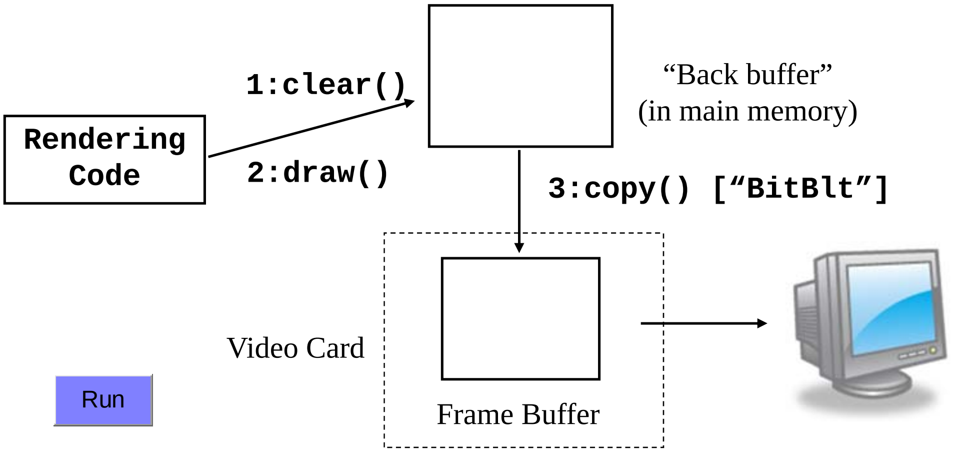
- Suppose the drawn output contains a triangle, continually changing location:



Run

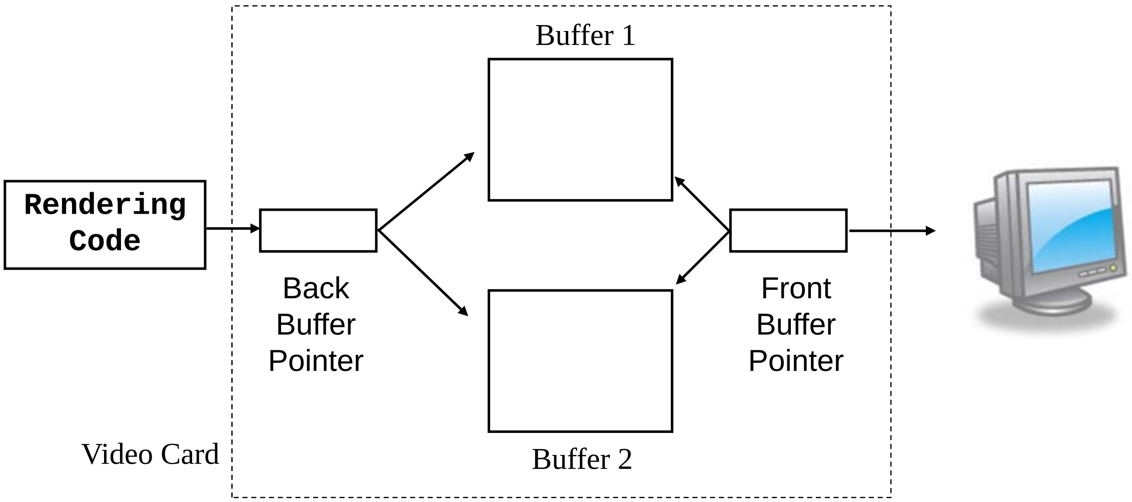
Double-Buffering

- Avoiding flicker:
  - Write to secondary or “back” buffer
  - Copy back buffer to “front” buffer when done



Page-Flipping

- Avoid BitBlt copy by changing a *pointer*



Buffer 1

Rendering Code

Back Buffer Pointer

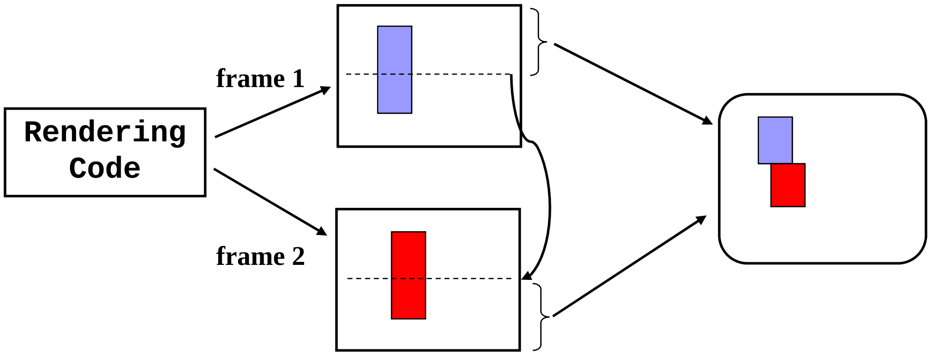
Front Buffer Pointer

Video Card

Buffer 2

Tearing

- Problem: swapping ½ way through scan
- Result: “torn image”
- Solution: hold off swap until “VSync”
  - Drawback: slows down renderer



frame 1

Rendering Code

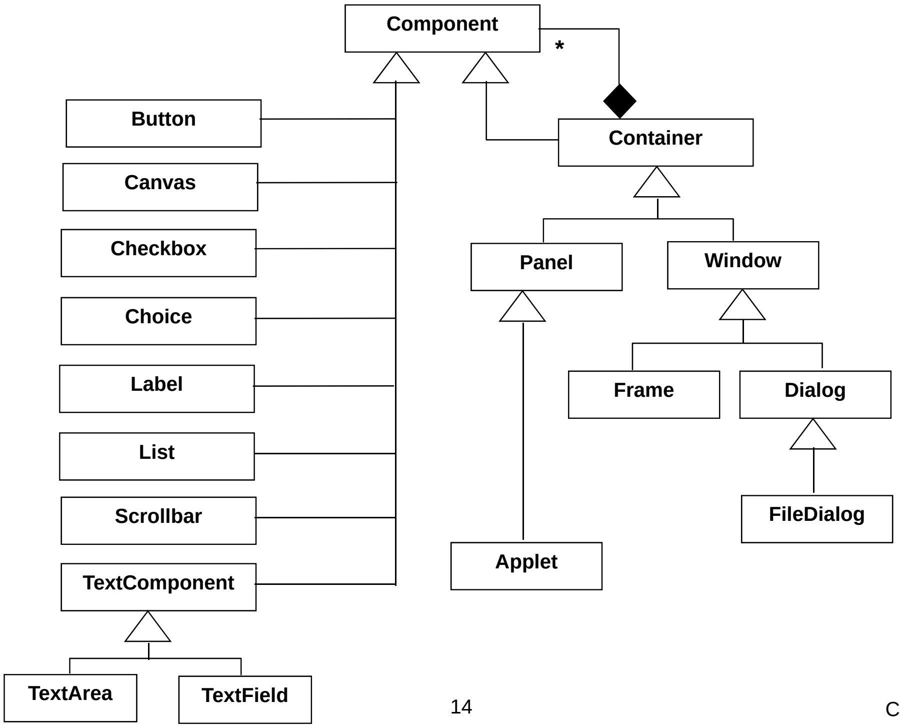
frame 2

*A*bstract *W*indowing *T*oolkit (AWT)

A *Framework* (a package named *java.awt*) consisting of:

- A *collection of reusable screen components*
  - "Component": an object having a *graphical representation*
  - Usually has the ability to *interact* with the user
  - Also called "widgets" (X-windows) or "controls" (Microsoft)
- An *event mechanism* connecting "actions" to "code"
- *Containers* and *Layout Managers* for arranging things on screen
- Support for *2D Graphics*
- Additional packages supporting applets, color, geometry, image processing, etc.

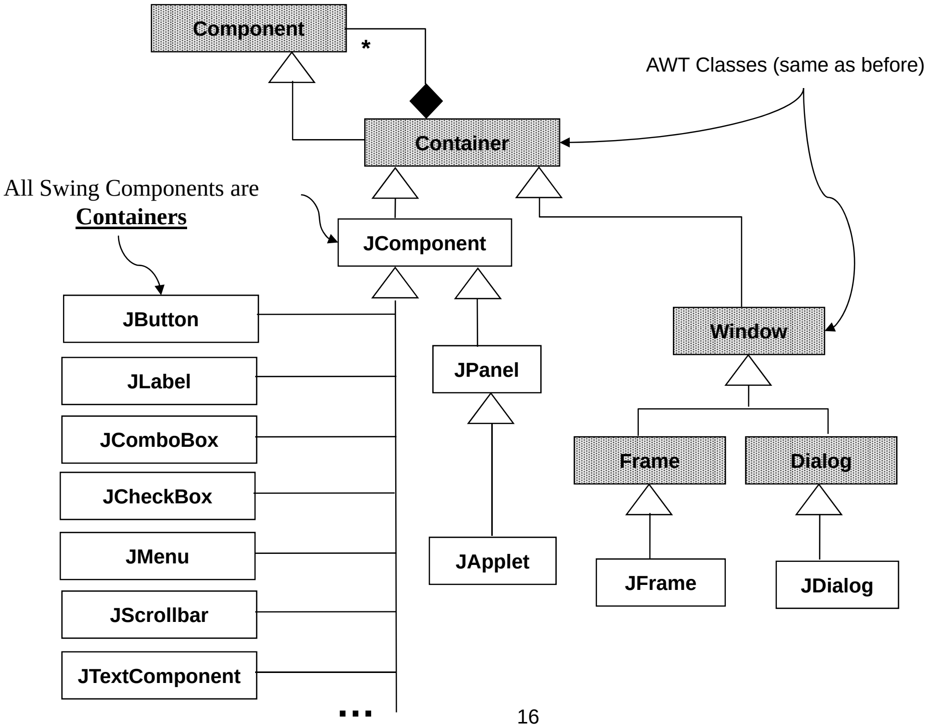
AWT Components (cont.)



JFC Facilities

- An extended set of GUI ***components:***
  - Top-level containers (JFrame, JApplet, JWindow, JDialog)
  - Basic Controls (JButton, JCheckBox, JRadioButton, JLabel, . . .)
  - Numerous other categories (displays, space-saving containers, …)
- A set of ***packages*** supporting a variety of capabilities:
  - "pluggable look-and-feel" (plaf)
  - borders
  - color choosers
  - event handlers
  - file choosers
  - text handlers (e.g. HTML and RTF)
  - tree manipulators
  - "undo" support

Swing Components



Creating A Frame

```
// Contents of File  DemoSimpleFrame.java:
/** This class is a driver for running the SimpleFrame class. It creates a JFrame
*/
public class DemoSimpleFrame {
  public static void main (String args[]) {
    //create one frame without a specified title
    SimpleFrame myFrame = new SimpleFrame ();
  }
}
==================================================

// Contents of File  SimpleFrame.java:
import javax.swing.JFrame;
/** This class creates a simple "Frame"  (a "window" that has certain
 *  adornments such as a title bar with buttons) by extending an existing
 *  class (in this case "JFrame", defined in the Java Swing package).
 */
public class SimpleFrame extends JFrame {
    // Construct a 200x100 frame and make it visible.
    public SimpleFrame () {
       setSize(200,100);           // specifies the size of the frame
       setVisible(true);           // tell window mgt system to show the frame
    }
}
```

Titled Frames

```
// Contents of File  DemoTitledFrame.java:
/** This class is a driver for demonstrating the TitledFrame class,
 *  which creates a simple "JFrame" with an optional Title Bar string.
 *  The driver creates two Frames, passing the command line parameter from
 *  the OS (if there was one) to the second constructor for use as the title.
 */
public class DemoTitledFrame {
  public static void main (String args[]) {
    String title = new String("Default Title");
    // get command-line title string if it was specified
    if (args.length > 0) {
      title = args[0] ;
    }
    // create one frame without a specified title
    TitledFrame myFrame1 = new TitledFrame ();
    // create a second frame with a specified title
    TitledFrame myFrame2 = new TitledFrame (title);
  }
}
```

Titled Frames (cont.)

```
// Contents of File  TitledFrame.java:
import javax.swing.JFrame;
/** This class creates a "Frame".  The constructors allow for the option
 *  of setting the Frame’s Title Bar text.
 */
public class TitledFrame extends JFrame {
  public TitledFrame () {
    // "no-arg" constructor
    setSize(200,100);
   setVisible(true);
  }
  public TitledFrame (String title) {
    // constructor initializing Title
    setTitle(title);
    setSize(300,200);
    setVisible(true);
  }
}
```

Frame Closing

```
// Contents of File DemoClosingFrame.java:
/** This class is a driver for demonstrating the
 *  ClosingFrame class, which changes the default
 *  "window closing" operation.
 */
public class DemoClosingFrame {
  public static void main (String args[]) {
      // create a frame with a specified title
      ClosingFrame myFrame = new ClosingFrame ("Exits on close");
  }
}
```

Frame Closing (cont.)

```
// Contents of File ClosingFrame.java:
import javax.swing.JFrame;
/** This class creates a frame with the default “window closing” operation
 *  changed so that the program will exit when the window frame is “closed”
 *  (the ‘X’ box is clicked), rather than being “hidden” (which is the default)
 *  The class also demonstrates how to position a frame on the screen.
 */
public class ClosingFrame extends JFrame {
  public ClosingFrame (String title) {
      setTitle (title);
      setSize (300,200);
      // tell window system where to position upper left corner of frame
      setLocation (200,200);
      // tell the frame what to do if 'X' is clicked
      setDefaultCloseOperation (EXIT_ON_CLOSE);
        // other choices include DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE
        // and HIDE_ON_CLOSE  (the default)
      setVisible(true);
  }
}
```

Adding Components to Frames

```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
/** This class defines a JFrame with several components in it. */
public class FrameWithComponents extends JFrame {
        public FrameWithComponents () {
                setTitle ("Frame With Components");
                setSize (250,150);
                setLocation (300,250);
                // create a new label object
                JLabel myLabel = new JLabel ("I'm A Label");
                // add the label to the "content pane" of the frame
                this.getContentPane().add(myLabel);
                // create a new labeled button and add it to the frame
                JButton doneButton = new JButton ("Push Me To
Quit");
                this.getContentPane().add(doneButton);
                // add a checkbox
                JCheckBox myCheckBox = new JCheckBox ("Reverse
Video",true);
                this.getContentPane().add(myCheckBox);
                // add a "combo box" (drop-down list)
                String [] items = {"banana", "orange",
"raspberry"};
                JComboBox myComboBox = new JComboBox (it
                this.getContentPane().add(myComboBox);
                this.getContentPane().setLayout(new
FlowLayout());
```

Run

Layout Managers

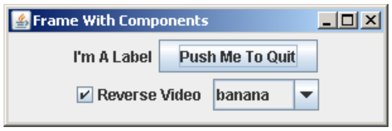
- Determine rules for positioning components in a container
  - Components which do not fit according to the rules may be hidden !!
- Layout Managers are classes
  - Must be instantiated and attached to their containers:

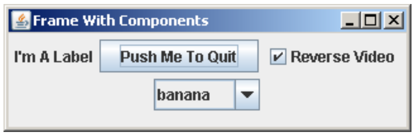
```
myContainer.setLayout( new FlowLayout() );
```

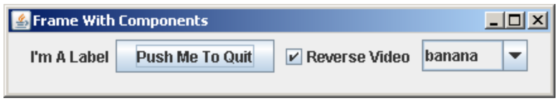
- Resizing a container *invokes the layout manager*
- Components have *size* attributes
  - MinimumSize, PreferredSize, and MaximumSize
  - Most components *set default sizes* when constructed
  - Layout managers *may or may not* respect sizes

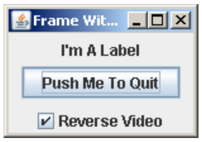
Layout Managers (cont.)

- Example: **`FlowLayout`**
  - Arranges components left-to-right, top-to-bottom (by default)
  - Components appear in the order they are added
  - Respects *preferred size*
  - Components that don't fit may be *hidden*







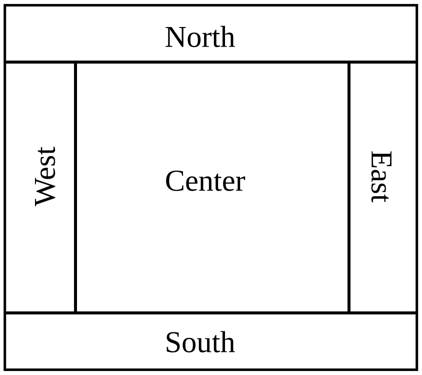


Run

Layout Managers (cont.)

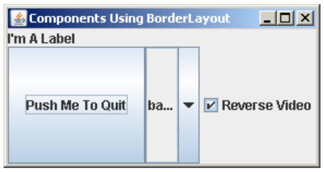
- Example: **`BorderLayout`**
  - Adds components to one of five “regions” of the container: North, South, East, West, or Center
  - Region must be specified when component is added

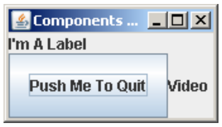
```
myContainer.add(theComponent,BorderLayout.NORTH);
```

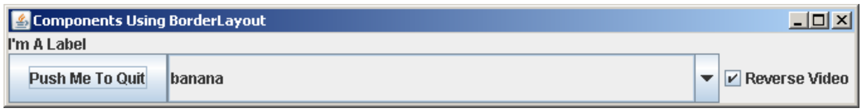


Layout Managers (cont.)

- **`BorderLayout`** (cont.)
  - Stretches North and South to fit, then East and West
    - Center gets what space is left (if any!)







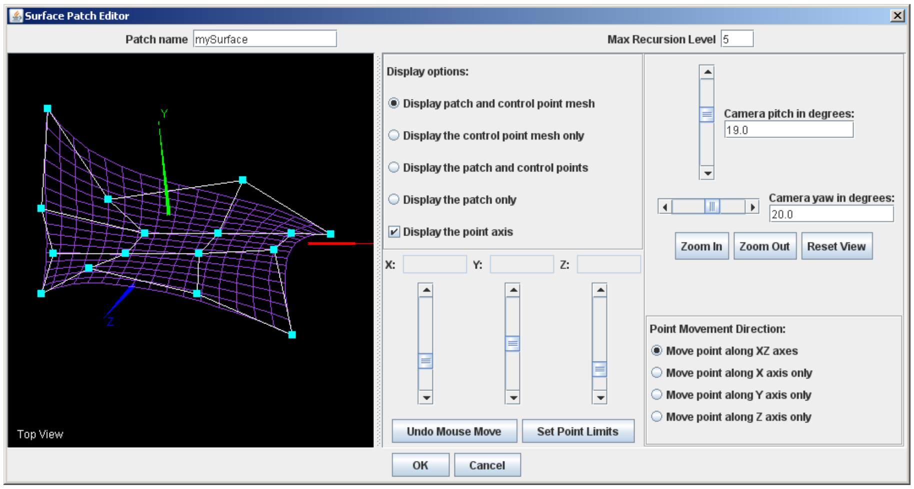
Run

Layout Managers (cont.)

- Other Layout Managers
  - `GridLayout`
  - `BoxLayout`
  - `CardLayout`
  - `GridBagLayout`
  - `SpringLayout`
  - `Custom`
- The *Strategy* Pattern

GUI Layout

GUIs usually have multiple “areas”



Surface Patch Editor

Patch name: mySurface

Max Recursion Level: 5

Display options:

● Display patch and control point mesh
○ Display the control point mesh only
○ Display the patch and control points
○ Display the patch only
☑ Display the point axis

Camera pitch in degrees: 19.0

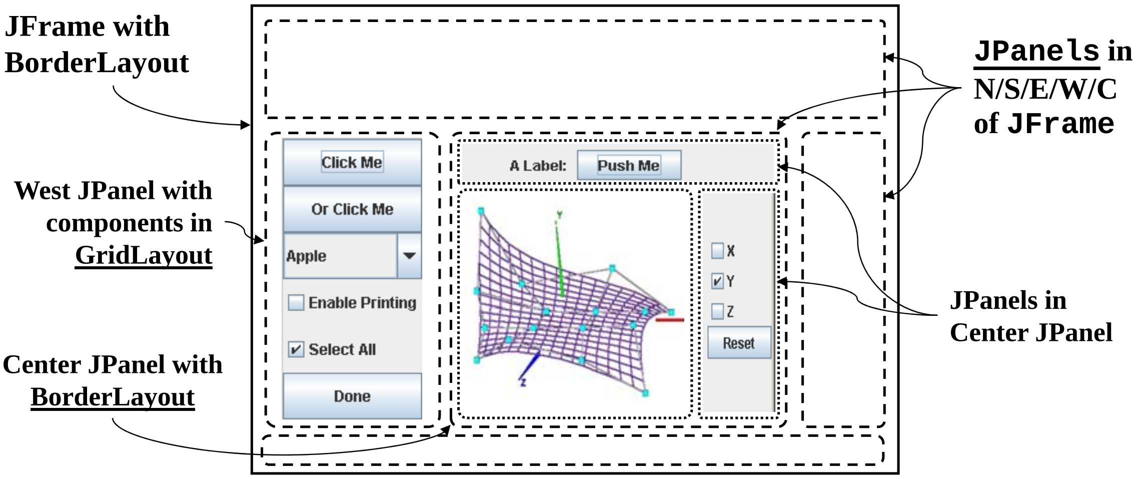
Camera yaw in degrees: 20.0

Point Movement Direction:

● Move point along XZ axes
○ Move point along X axis only
○ Move point along Y axis only
○ Move point along Z axis only

The JPanel Class

- **JPanel**: an *invisible* component that…
  - Can be assigned to an area
  - Can have a layout manager assigned to it



JPanel Example

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
// A JFrame with panels in different layout arrangements
public class PanelLayoutFrame extends JFrame {

  public PanelLayoutFrame () {
    // code here to initialize the Frame as before...

    // add a bordered panel at the top (North) of this frame (the frame’s
    // ContentPane already has a BorderLayout manager by default)
    JPanel topPanel = new JPanel();
    topPanel.setBorder (new LineBorder(Color.blue,2));
    this.add(topPanel,BorderLayout.NORTH);

    // add a label and a button to the top panel; they will lay out centered
    // from left to right due to the default operation of the default Layout
    // Manager for JPanel (FlowLayout)
    JLabel myLabel = new JLabel ("Label:") ;
    topPanel.add(myLabel);
    JButton mytopButton = new JButton ("Push Me");
    topPanel.add(mytopButton);

    // add a bordered panel with a 10x1 Grid Layout at the left of this frame
    JPanel leftPanel = new JPanel();
    leftPanel.setBorder (new TitledBorder(“ Options: “));
    leftPanel.setLayout (new GridLayout (10,1));
    this.add(leftPanel,BorderLayout.WEST);

    ... continued
```

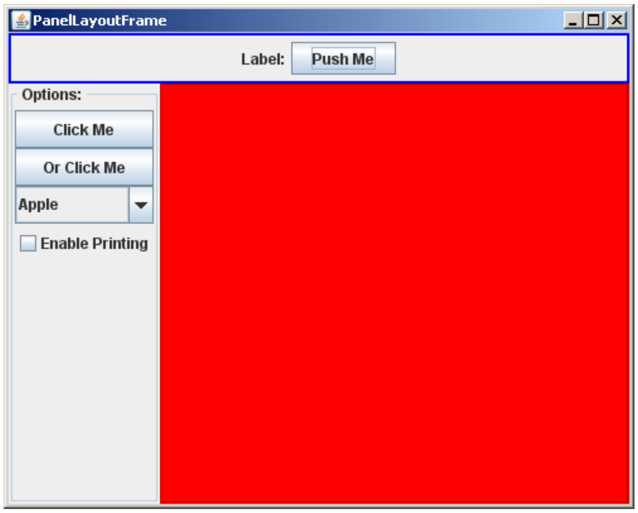
JPanel Example (cont.)

```
    ... continued

    // add two buttons to the left panel; they will be in the top two grid cells
    JButton myLeftButton1 = new JButton ("Click Me");
    leftPanel.add(myLeftButton1);
    JButton myLeftButton2 = new JButton ("Or Click Me");
    leftPanel.add(myLeftButton2);

    // add a "combo box" (drop-down list) with a list of fruits to the left panel
    String [] fruitList = new String [] {"Apple", "Orange", "Banana"} ;
    JComboBox fruitComboBox = new JComboBox (fruitList);
    leftPanel.add(fruitComboBox);

    // add a "check box" to the left panel
    JCheckBox myCheckBox = new JCheckBox ("Enable Printing");
    leftPanel.add(myCheckBox);

    // add a colored bordered panel at the center of this frame
    JPanel centerPanel = new JPanel();
    centerPanel.setBorder (new EtchedBorder());
    centerPanel.setBackground (Color.red);
    this.add(centerPanel,BorderLayout.CENTER);

    setVisible(true);
  }
}
```

Run

JPanel Example – Output

JPanel in North with LineBorder containing Label and Button

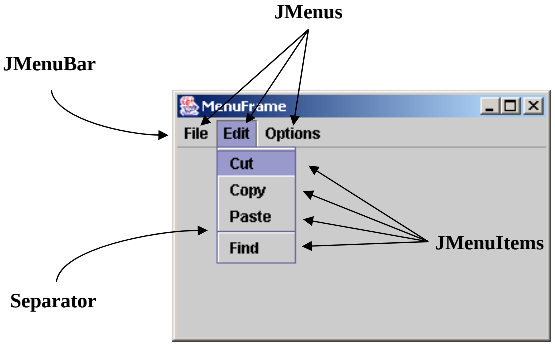


JPanel in West with TitledBorder containing several components in a 10x1 grid

JPanel in Center with EtchedBorder and red background

Menus

- **`JMenuBar :`** attached to a **`JContentPane`**
- **`JMenuBars`** can hold **“`JMenu`”** components
- Each **`JMenu`** can hold **“`JMenuItem`”** components
- Menu items can have “separators” between them



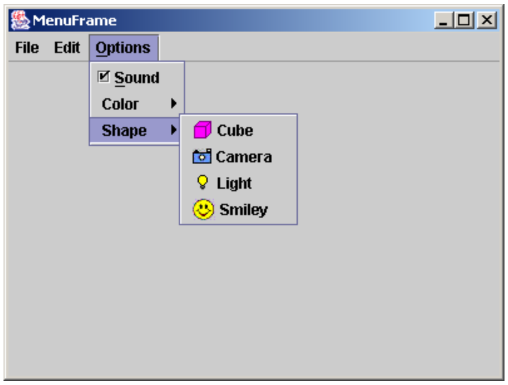
Menu Building Example

```
/** This class defines a JFrame with a Menu Bar containing several menu items */
public class MenuFrame extends JFrame {
  public MenuFrame () {
    // ... code here to initialize the frame (size, location, etc.) ...
    JMenuBar bar = createMenuBar();
    this.setJMenuBar(bar);
    this.setVisible(true);
   }
  private JMenuBar createMenuBar() {
    JMenuBar bar = new JMenuBar();          //create the menu bar object
    // ... code here to create a FILE Menu and add it to the menu bar ...
    // The following code creates an EDIT Menu and adds it to the menu bar:
    JMenu editMenu = new JMenu("Edit");
      // create each of the Edit menu items and add each item to the menu
      JMenuItem mItem = new JMenuItem("Cut");       // edit->cut
      editMenu.add(mItem);
      mItem = new JMenuItem("Copy");                // edit->copy
      editMenu.add(mItem);
      mItem = new JMenuItem("Paste");               // edit->paste
      editMenu.add(mItem);
      editMenu.addSeparator();                      // adds a separator
      mItem = new JMenuItem("Find");                // edit->find
      editMenu.add(mItem);
      bar.add(editMenu);   // add Edit menu to menu bar
    // ... code here to add OPTIONS Menu to the menu bar ...
    return bar;
  }
}
```

Run

Complex Menus

- **Menus and Menu Items can contain a variety of things:**
  - Submenus
  - Components
  - Icons
  - Mnemonic key bindings
  - Accelerator key bindings



Complex Menu Example

```
public class MenuFrame extends JFrame {
  ...
  private JMenuBar createMenuBar () {
    JMenuBar bar = new JMenuBar();
    JMenuItem menuItem ;
    // ... code here to create FILE and EDIT menus as before ...
    // create an OPTIONS Menu
    JMenu optionsMenu = new JMenu("Options");
      optionsMenu.setMnemonic ('O');
      // create a Sound menu with On/Off checkbox
      JCheckBoxMenuItem soundMenu = new JCheckBoxMenuItem( "Sound" );
      soundMenu.setMnemonic('S');
      optionsMenu.add( soundMenu );
      // create a Color menu with colored entries on a submenu
      JMenu colorMenu = new JMenu( "Color" );
      JMenuItem redMenuItem = new JMenuItem("red");
      redMenuItem.setBackground (new Color(255,0,0));
      colorMenu.add(redMenuItem);
      JMenuItem greenMenuItem = new JMenuItem("green");
      greenMenuItem.setBackground (new Color(0,255,0));
      colorMenu.add(greenMenuItem);
      // ... add other color menuItems here ...
      optionsMenu.add( colorMenu );
      ...continued
```

Complex Menu Example (cont.)

```
    ...continued
    // create a Shape menu with Icons
    JMenu imageMenu = new JMenu( "Shape" );
    JMenuItem cube = new JMenuItem("Cube", new ImageIcon("Cube.gif"));
    imageMenu.add(cube);
    JMenuItem camera = new JMenuItem("Camera", new ImageIcon("Camera.jpg"));
    imageMenu.add(camera);
    JMenuItem light = new JMenuItem("Light", new ImageIcon("Light.gif"));
    imageMenu.add(light);
    JMenuItem smiley = new JMenuItem("Smiley", new ImageIcon("HappyFace.gif"));
    imageMenu.add(smiley);
    optionsMenu.add( imageMenu );
    bar.add(optionsMenu);
   return bar ;
  }
}
```

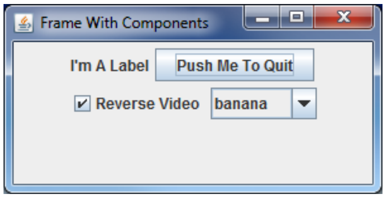
Run

Programmable Look-And-Feel

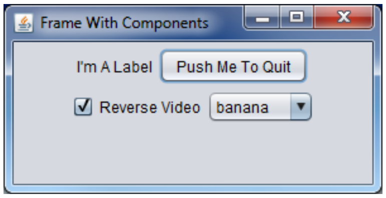
- Swing components are MVC-based
- “View” determines how components “look”
  - Windows look
  - Unix (X-windows) look
  - MacOS look
  - Java-specific look
- Program code can *change* the “View”, known as the “Look And Feel” (LAF):

```
UIManager.setLookAndFeel(
       UIManager.getSystemLookAndFeelClassName() );
```

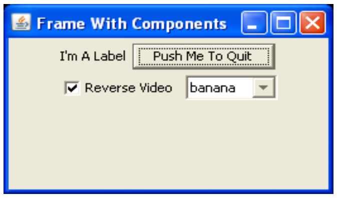
PLAF Examples



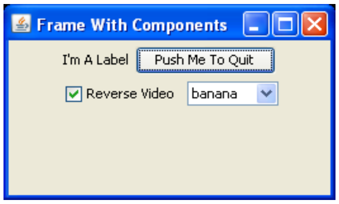
Java “Metal” LAF



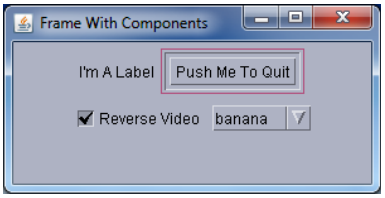
Java “Nimbus” LAF



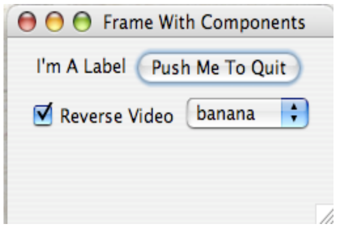
Windows “Classic” LAF



Windows XP LAF



CDE/Motif (Solaris) LAF



Mac OS/X LAF

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

IX – Event-Driven Programming

- **Traditional vs. Event-Driven Programming**
- **Event Objects**
- **Event Listeners**
- **Java AWT Events and Listeners**
- **Action Events and Listeners**
- **Key Events and Key Bindings**
- **Mouse Events and Listeners**
- **Window Events and Listeners**
- **Component Events and Listeners**
- **Item Events and Listeners**

CSc 133 Lecture Notes

IX - Event-Driven Programming

Computer Science Dept.
CSUS

Overview

- Traditional vs. Event-Driven Programs
- Events and Event-Listeners
- AWT Event Types
  - Action, Window, Component, Item, Mouse, and Key Events
- Key Bindings

Traditional vs. Event-Driven

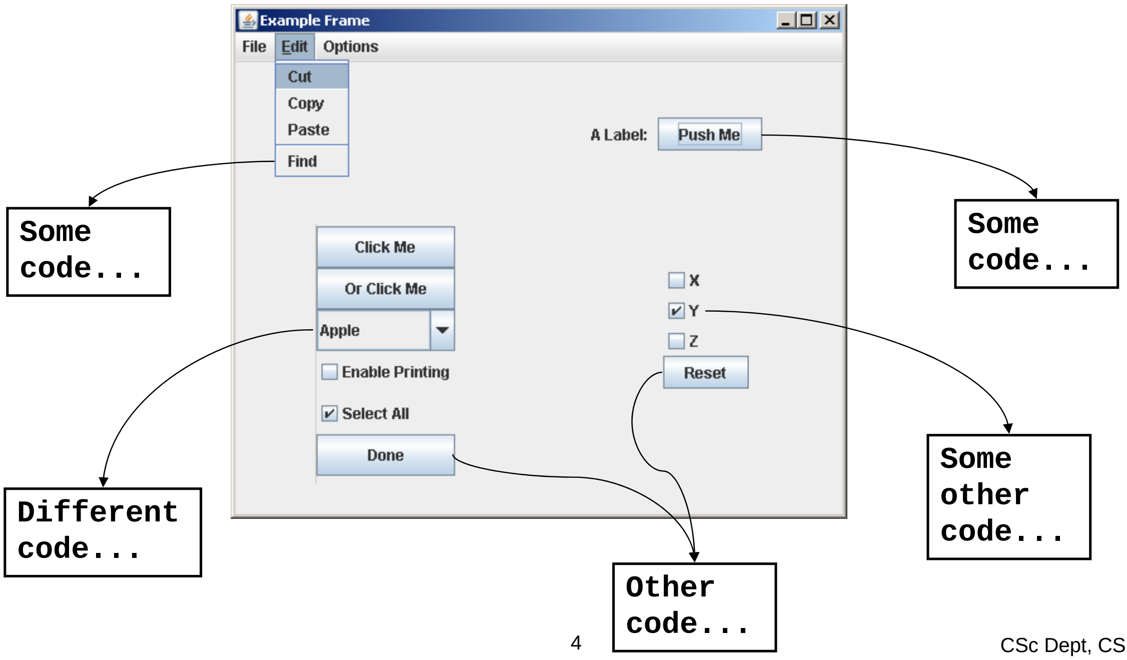
- Traditional program organization:

```
main: {
     loop {
          get some input ;
          process input ;
          produce output ;
     }
     until (done);
}
```

- Event-driven program organization:

```
main: {
     make a GUI frame ;
     make some controls (buttons, etc.) ;
     use a layout manager to position controls ;
     make the frame visible ;
}
```

Event-Driven Operation



Some code...

Some code...

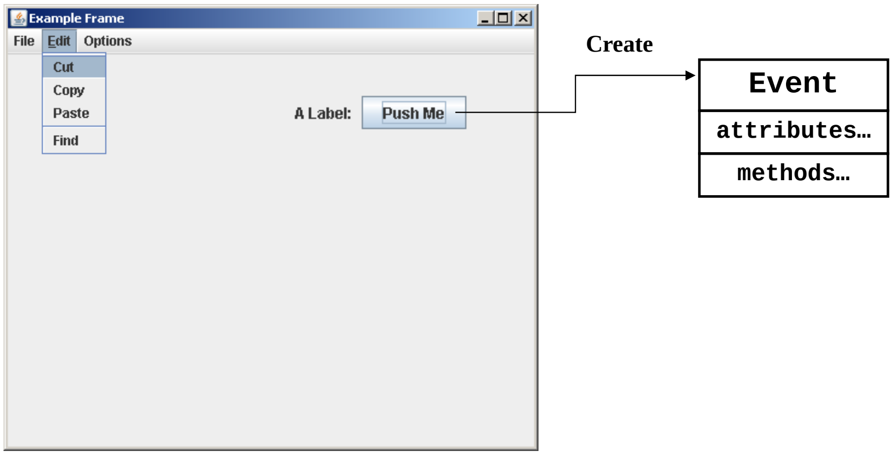
Different code...

Some other code...

Other code...

Event Objects

- Activating a component *creates an object of type "Event"*



Create

Event

attributes…

methods…

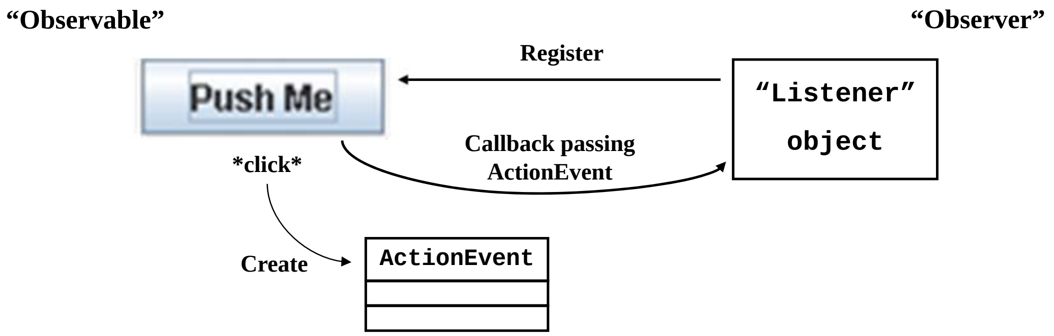
AWT Events

Event Name	Can Be Created By
`ActionEvent`	Pushing a button; selecting a menu item; selecting a drop-down list item; …
`ItemEvent`	Checking/unchecking a checkbox; selecting/unselecting a "radio button"; …
`WindowEvent`	Closing, minimizing, maximizing, opening, … a window
`ComponentEvent`	Resizing a window, ….
`TextEvent`	Pushing newline (enter) key in a text field; …
`MouseEvent`	Pressing or releasing a mouse button; …
`AdjustmentEvent`	Moving a scrollbar; …
`KeyEvent`	Pressing a keyboard key
`MouseWheelEvent`	Rotating mouse scroll wheel

Event Listeners

- Event-driven code attaches *listeners* to *event-generators*

- Event-generators make *call-backs* to listeners



“Observable”

“Observer”

Push Me

Register

“Listener” object

Callback passing ActionEvent

click

Create

ActionEvent

ActionListener Interface

- **Listeners for *ActionEvents* must implement interface *ActionListener* :**

```
interface ActionListener
{
  public void actionPerformed (ActionEvent e);
}
```

Creating An ActionListener

```
import java.awt.event.*;

/** This class acts as a listener for ActionEvents.
 *  It was designed to be attached and respond
 *  to button-push events.
 */
public class ButtonListener implements ActionListener
{
  // Action Listener method:  called from the object being observed
  // (e.g. a  button) when it generates an "Action Event"
  // (which is what a button-click does)

  public void actionPerformed (ActionEvent e)
  {
     // we get here because the object being observed
     // generated an Action Event
     System.out.println ("Button Pushed...");
  }
}
```

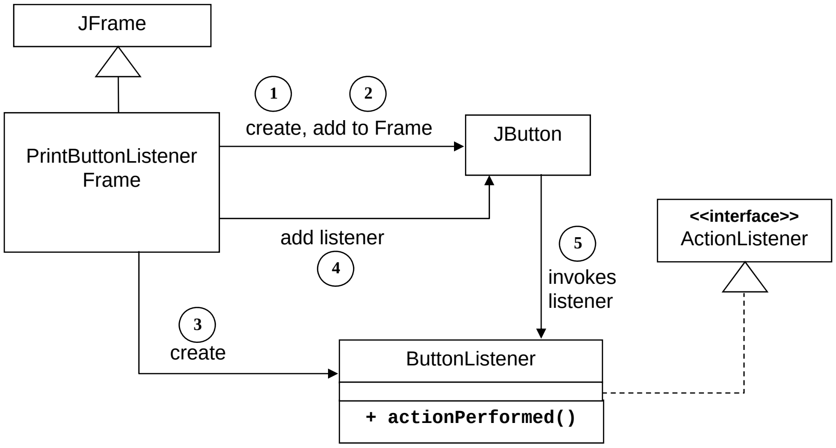
Using An ActionListener

```
import java.awt.*;
import javax.swing.*;

/** This class is a JFrame with a single Button to which is attached an
 *  ActionListener. The button action listener is invoked whenever the
 *  button is pushed.
 */
public class PrintButtonListenerFrame extends JFrame {

  public PrintButtonListenerFrame () {

     //...code here to initialize the frame (size, position,
     //   layout manager, etc. as usual...

     //create a new button
     JButton printButton = new JButton ("Print");

     //add the button to the content pane of this object (the frame),
     //  positioned as determined by the layout manager
     this.add(printButton);

     //create a separate ActionListener for the button
     ButtonListener pbListener = new ButtonListener();

     //register the pbListener as an Action Listener for
     //  action events from the button
     printButton.addActionListener(pbListener);

     setVisible(true);
  }
}
```

Run

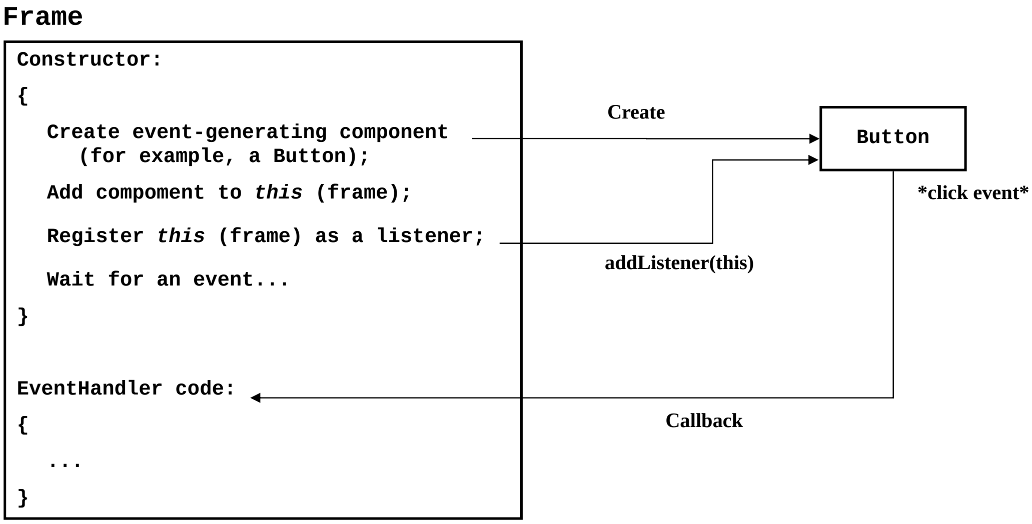
Listener Class Organization

- UML for the previous code:



Alternative Organization

Frames can listen to their own components!



```
Frame
Constructor:
{
   Create event-generating component
      (for example, a Button);
   Add compoment to this (frame);
   Register this (frame) as a listener;
   Wait for an event...
}

EventHandler code:
{
   ...
}
```

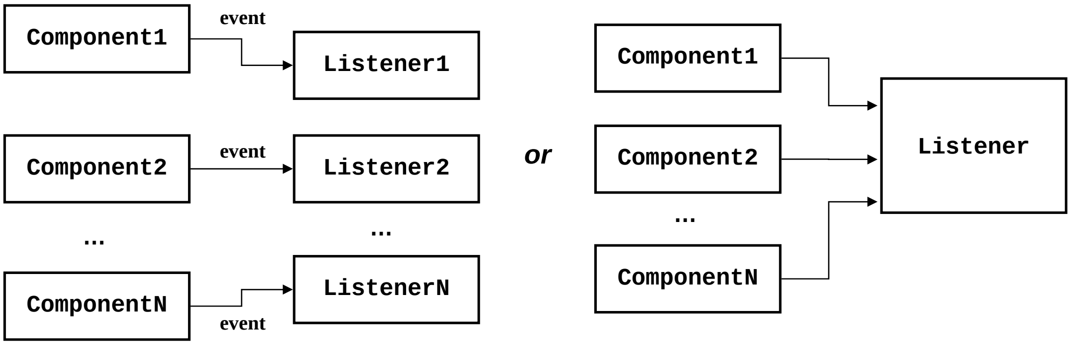
ActionListener Frame Example

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/** A JFrame with a single button which the Frame listens to. */
public class SelfListenerFrame extends JFrame implements ActionListener {
  public SelfListenerFrame () {
    // ...code here to initialize the frame (size, location, layout, etc.)
    // create a new button
     JButton doneButton = new JButton ("Push Me");
    // add the button to the content pane of this frame
    this.add(doneButton);
    // register THIS object (the frame) as an Action Listener for
    //  action events from the button
     doneButton.addActionListener(this);
     setVisible(true);
  }
  // Action Listener method: called from the button because
  // this object -- the frame -- is an action listener for the button
  public void actionPerformed (ActionEvent e) {
     System.out.println ("Frame: button pushed");
  }
}
```

Run

Multiple Event Sources

- *Two approaches:*
  - Multiple separate listeners
  - One listener
    - it would need to be able to *distinguish event source*



Let's consider this second option …

```
/** A JFrame with a ComboBox and buttons, with action handlers in the frame */
public class MultipleComponentListener extends JFrame implements ActionListener
{
  private JButton xButton, yButton, quitButton ; // the frame's buttons
  private JComboBox myComboBox ;                 // the frame's drop-down list

  public MultipleComponentListener () {
    // ... code here to initialize the Frame as usual ...

    // create buttons and add them to this object (the frame)
    xButton = new JButton ("Print X's");
    this.add(xButton);
    yButton = new JButton ("Print Y's");
    this.add(yButton);
    quitButton = new JButton ("Quit");
    this.add(quitButton);

    // create and add a "combo box" (drop-down list)
    myComboBox = new JComboBox () ;
    myComboBox.addItem (new String ("Lemon") ) ;
    myComboBox.addItem (new String ("Lime") ) ;
    myComboBox.addItem (new String ("Grapefruit") ) ;
    this.add(myComboBox) ;

    // register this object (the frame) as an Action Listener for
    // action events from buttons and the Combo Box
    xButton.addActionListener(this);
    yButton.addActionListener(this);
    quitButton.addActionListener(this);
    myComboBox.addActionListener(this);

    setVisible(true);
  }
```

```
  // ...continued
  // Action Listener method:  called when any component generates an ActionEvent
  public void actionPerformed (ActionEvent e) {
    // Check the contents of the received ActionEvent" object to see who generated it.
    // One way to check: look at the "Action Command" (normally the LABEL):
    if (e.getActionCommand().equals("Print X's"))
    {
      System.out.println ("XxXxXxXxXxXxXxXx");
    }
    // another way to check:  see what OBJECT generated the event
    else if (e.getSource() == yButton)
    {
      System.out.println ("yYyYyYyYyYyYyYyY");
    }
    else if (e.getSource() == myComboBox)
    {
      System.out.println("Current selection:" + myComboBox.getSelectedItem());
    }
    else if (e.getSource() == quitButton)
    {
      System.exit(0);    //really should confirm intent before exiting…
    }
  } //end actionPerformed()
} //end frame
```

Run

Example using Menus

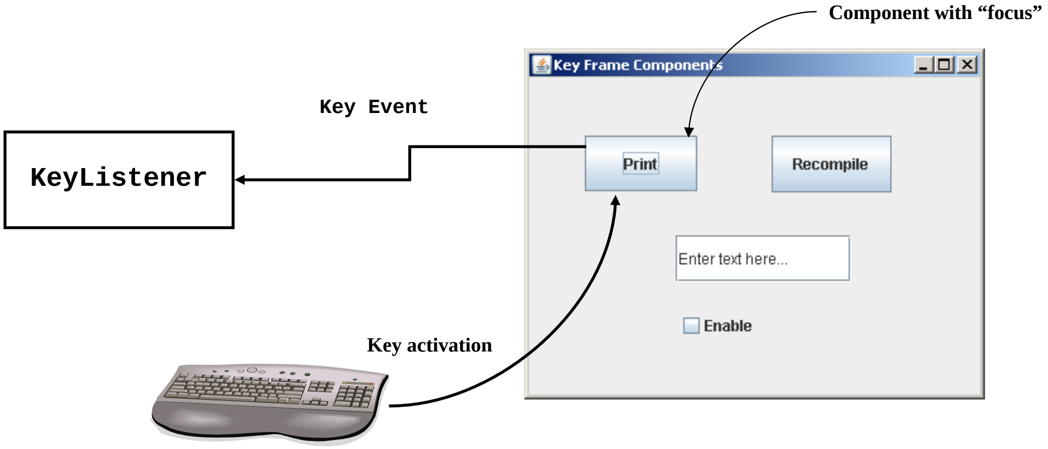
```
/** A JFrame with a Menu, and action handlers for menu items. */
public class MenuFrame extends JFrame implements ActionListener {
  public MenuFrame () {
    //...code here to initialize the frame as usual...
    setJMenuBar(createMenuBar());
    setVisible(true);
  }
  private JMenuBar createMenuBar () {
    JMenuBar bar = new JMenuBar();
    JMenu fileMenu = new JMenu("File");
    // the frame is the listener for each menu item
    JMenuItem mItem = new JMenuItem("Open");       //file->open
    mItem.addActionListener(this);
    fileMenu.add(mItem);
    mItem = new JMenuItem("Save");                 //file->save
    mItem.addActionListener(this);
    fileMenu.add(mItem);
    fileMenu.addSeparator();
    mItem = new JMenuItem("Quit");                 //file->quit
    mItem.addActionListener(this);
    fileMenu.add(mItem);
    bar.add(fileMenu);
    //code here to create other menus and add listeners to them ...
    //add the menu bar to the frame
    return bar;
  }
  //...continued
```

```
//...continued...
public void actionPerformed (ActionEvent e) {
  // Same approach as before, to show how this organization can get ugly
  if (e.getActionCommand().equals("Open")) {
    FileDialog fdlg = new FileDialog (this, "Open File");
    fdlg.setVisible(true);                     // note that dialog is "modal"
    String fileName = fdlg.getFile();          // get user-selected file name
    if (fileName != null) {
      System.out.println ("File '" + fileName + "' selected for opening");
      // add code here to actually OPEN the file...
    } else {
      System.out.println ("No file name selected");
    }
  } else if (e.getActionCommand().equals("Save")) {
    System.out.println ("File->Save was selected") ;
  } else if (e.getActionCommand().equals("Quit")) {
    System.out.println ("ByeBye");
    System.exit(0);            // quits without confirmation – poor!
  } else if (e.getActionCommand().equals("Copy")) {
    System.out.println ("Edit->Copy was selected") ;
  } else if (e.getActionCommand().equals("Paste")) {
    System.out.println ("Edit->Paste was selected") ;
  }
  // ... action handlers for other menu items here ...
} //end actionPerformed()
} //end frame
```

Run

Key Events

- Generated by the ***component with “focus”*** when a key is “activated”
- Handled by KeyListener objects



Component with “focus”

Key Event

KeyListener

Key activation

Key Event Listeners

- KeyListeners must implement

```
public interface KeyListener {
    keyPressed (KeyEvent e);
    keyReleased (KeyEvent e);
    keyTyped (KeyEvent e);
}
```

- KeyEvents contain “Virtual Keycodes”
  - Defined in class **KeyEvent**
  - Identify the physical key that was “activated”

```
/** A JFrame with a Button and a keyboard listener which changes the button label.*/
public class KeyListenerFrame extends JFrame implements KeyListener {
  public KeyListenerFrame () {
    // ...code here to initialize the frame
    JButton printButton = new JButton ("Print");   // add a button to the frame
    this.add(printButton);
    // make the frame a listener for keystrokes sent to the button
    printButton.addKeyListener (this);
    setVisible(true);
  }
  public void keyPressed (KeyEvent event) {
    switch (event.getKeyCode()) {
      case KeyEvent.VK_A:
        ((JButton)event.getSource()).setText("A New Label"); break;
      case KeyEvent.VK_P:
        ((JButton)event.getSource()).setText("Print"); break;
      case KeyEvent.VK_UP:
        ((JButton)event.getSource()).setText("Up"); break;
      default:
        System.out.println ("Key: '" + event.getKeyChar() + "'");
    }
  }
  // other (empty) keylistener methods
  public void keyTyped (KeyEvent event) {}
  public void keyReleased (KeyEvent event) {}
}
```

Run

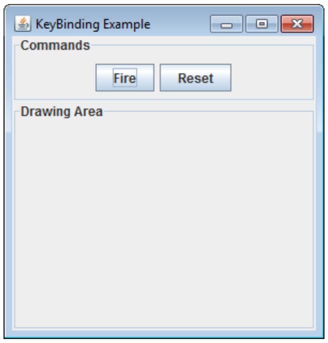
KeyListener Issues

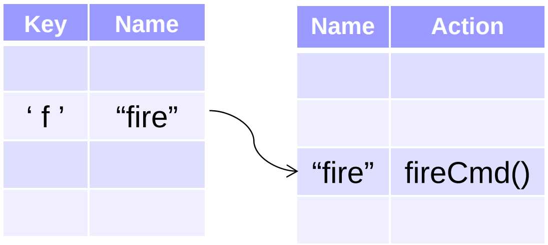
- **KeyEvent only forwarded to Listeners under certain conditions**
  - **Related to which component has "focus"**
  - **Focus rules are complex**
    - **…and vary between Java versions…**
  - **Focus can be *changed* by different actions**
    - **Mouse clicks**
    - **Tab character**
    - **Application code**

Run

Key Bindings

- Every `JComponent` has a set of
  `Key→Action` *maps ("bindings")*
- Two kinds of maps
  - Input maps (hold `KeyStroke` objects)
  - Action maps (hold `Action` objects)





Key	Name
' f '	"fire"

Input Map for GUI
"Fire" button

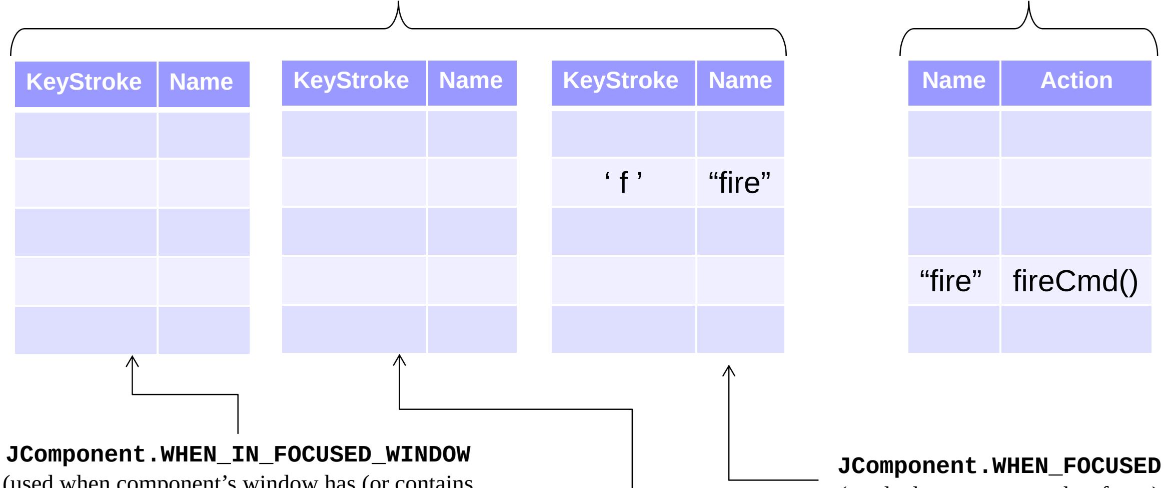
Name	Action
"fire"	fireCmd()

Action Map for GUI
"Fire" button

Input / Action Maps

Input Maps (one 3-map set for each JComponent)

Action Map (one per JComponent)



KeyStroke	Name

KeyStroke	Name

KeyStroke	Name
' f '	"fire"

Name	Action
"fire"	fireCmd()

`JComponent.WHEN_IN_FOCUSED_WINDOW`
(used when component's window has (or contains the component which has) focus)

`JComponent.WHEN_FOCUSED`
(used when component has focus)

`JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT` (used when component *contains* the component which has focus)

Key Bindings (cont.)

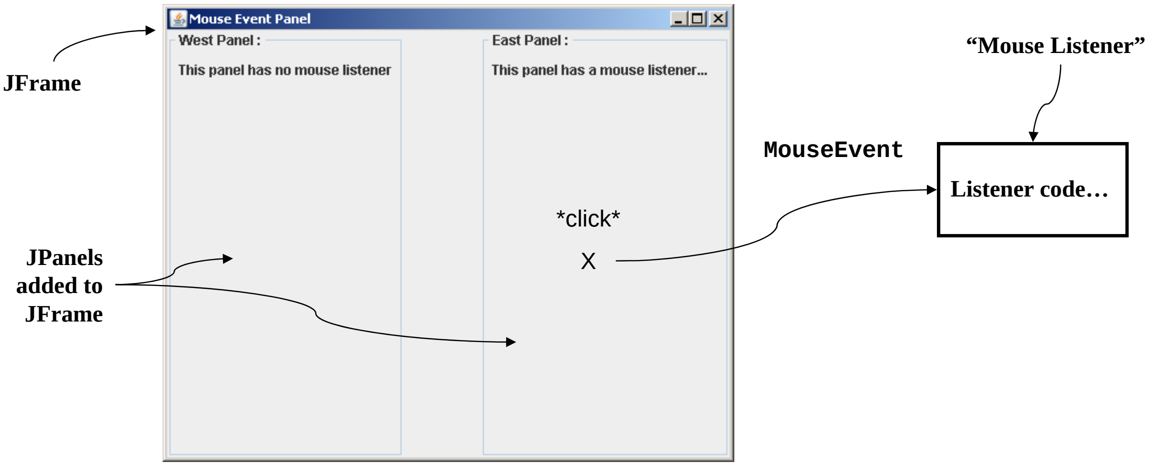
- **Setting up a key binding**
  - **Create a `KeyStroke` object**
  - **Store `KeyStroke` in component's Input Map under some "name"**
    - **Typically, command name is used**
  - **Create an `Action` object**
  - **Store `Action` in component's Action Map under same name**

```
/** This code snippet uses key bindings so the 'f' key invokes an Action object
 *  (command) which is also associated with a button. Once this code is executed,
 *  pressing the 'f' key or pressing the "Fire" button will invoke the "fireCommand"
 *  object any time the focus lies anywhere in the window.
 */
// create a "FireCommand" object
FireCommand fireCommand = new FireCommand();

// add the firecommand as a button action
JButton fireButton = new JButton ("Fire");
fireButton.setAction(fireCommand);
...
// get the "focus is in the window" input map for the center panel
int mapName = JComponent.WHEN_IN_FOCUSED_WINDOW;
InputMap imap = centerPanel.getInputMap(mapName);
// create a keystroke object to represent the "f" key
KeyStroke fKey = KeyStroke.getKeyStroke('f');
// put the "f-Key" keystroke object into the panel's "when focus is
// in the window" input map under the identifier name "fire"
imap.put(fKey, "fire");   //hashtable(k,v)
// get the action map for the panel
ActionMap amap = centerPanel.getActionMap();
// put the "fireCommand" object into the panel's actionMap
amap.put("fire", fireCommand);     //hashtable(k,v)
//have the frame request keyboard focus
this.requestFocus();
```

Run

Mouse Events

- Certain components (e.g. **`JPanels`**) can generate **`MouseEvents`**



Mouse Event Panel

West Panel :

This panel has no mouse listener

East Panel :

This panel has a mouse listener...

Mouse Events (cont.)

- **Generated when a mouse *button* is <u>clicked</u>, <u>pressed</u>, or <u>released</u>**
  - Click = press then release *without moving* the mouse
  - Also generated when the mouse "enters" or "exits" a component
- **<u>Mouse Clicked</u> implies Mouse Pressed and Mouse Released**
  - Mouse Pressed event occurs first
  - Order of Clicked/Released is implementation-dependent

MouseListeners

- **MouseListener objects must implement**

```
public interface MouseListener {
   public void mouseClicked (MouseEvent e) ;
   public void mousePressed (MouseEvent e);
   public void mouseReleased (MouseEvent e);
   public void mouseEntered (MouseEvent e) ;
   public void mouseExited (MouseEvent e) ;
}
```

- **MouseListeners can be**
  - Separate classes
  - Components also serving other purposes (e.g. Frames)

MouseListener Example

```
/** A JFrame with a simple mouse-responding panel */
public class MouseEventPanel extends JFrame implements MouseListener {
  private JPanel myLeftPanel, myRightPanel ;
  public MouseEventPanel() {
    // ...code here to initialize the frame (size, location, layout, etc.)
    // create a panel with no mouse listener
    myLeftPanel = new JPanel();
    myLeftPanel.setBorder (new TitledBorder(" West Panel : "));
    myLeftPanel.add (new JLabel("This panel has no mouse listener"));
    add (myLeftPanel, BorderLayout.WEST);
    // create a panel with "this" (frame) as its mouselistener
    myRightPanel = new JPanel();
    myRightPanel.setBorder (new TitledBorder(" East Panel : "));
    myRightPanel.add (new JLabel("This panel has a mouse listener..."));
    myRightPanel.addMouseListener (this);
    add (myRightPanel, BorderLayout.EAST);
    ...
  }
  public void mousePressed (MouseEvent e) {
    if (e.getButton() == MouseEvent.BUTTON1)
      System.out.println (“Left Mouse Button pressed...");
  }
  public void mouseClicked (MouseEvent e) { }
  public void mouseReleased (MouseEvent e) { }
  public void mouseEntered (MouseEvent e) { }
  public void mouseExited (MouseEvent e) { }
}
```

Run

MouseListener Example (2)

```
/** Mouse PRESSED toggles between two colors; mouse CLICKED
 *  (press/release) with no motion) resets to a default color.
*/
public class ColorToggle extends JFrame implements MouseListener {
  private JPanel myPanel ;
  private Color defaultColor=Color.green, currentColor=Color.red,
                nextColor=Color.blue;
  public ColorToggle() {
    //...code here to initialize the Frame...
    //create a center panel which will change colors on mouse events
    myPanel = new JPanel();
    myPanel.setSize (300,300);
    myPanel.setBackground (defaultColor);
    myPanel.addMouseListener (this);
    add (myPanel, BorderLayout.CENTER);
    this.setVisible (true);
  }
  //...continued
```

MouseListener Example (2) (cont.)

```
  // ...ColorToggle class continued...
  public void mousePressed (MouseEvent e) {
    // swap the alternating colors
    Color temp = currentColor;
    currentColor = nextColor;
    nextColor = temp;
    // redraw the panel with a new background color
    myPanel.setBackground(currentColor);
  }

  public void mouseClicked (MouseEvent e) {
    myPanel.setBackground (defaultColor);
  }

  public void mouseReleased (MouseEvent e) { }

  public void mouseEntered (MouseEvent e) {
    System.out.println ("Mouse entered panel") ;
  }

  public void mouseExited (MouseEvent e) {
    System.out.println ("Mouse exited panel") ;
  }
} // end ColorToggle class
```

Run

Mouse Wheel Events

- **Generated when wheel is rotated in a component**
- **Handled by MouseWheelListener objects**

```
public interface MouseWheelListener {
    public void mouseWheelMoved (MouseWheelEvent e);
}
```

- **Attaching MouseWheelListeners to components:**

```
theComponent.addMouseWheelListener (aWheelListener);
```

- **Determining amount of wheel movement:**

```
//returns the number of “clicks” the mouse wheel was rotated
int numClicks = event.getWheelRotation();
```

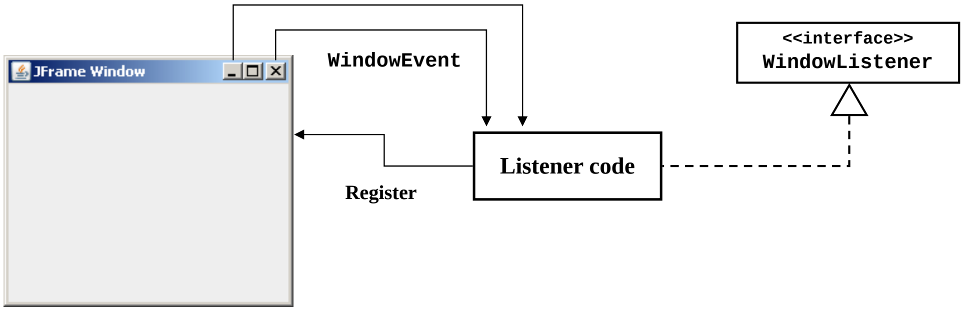
- **MouseWheelEvents can be forwarded to “parent container” components (e.g. ScrollPanes)**
  - Certain components support various units of movement due to mouse wheel motion**:** Text lines, pages, other blocks

Window Events

- **Generated (“fired”) when a `Window` is**
  - **Opened**
  - **Closing (user has requested a close)**
  - **Closed**
  - **Iconified (“minimized”)**
  - **De-iconified (“restored”)**
  - **Activated (receives keyboard focus; becomes the active window)**
  - **Deactivated**
- **Handled by `WindowListener` objects**

Window Listeners

- Any component can be a “WindowListener”
  - Register with the window (e.g. **`JFrame`**) using **`addWindowListener()`**
  - Implement interface **`WindowListener`**



WindowListener Interface

```
public interface WindowListener
{
   public void windowOpened (WindowEvent e) ;
   public void windowClosing (WindowEvent e) ;
   public void windowClosed (WindowEvent e) ;
   public void windowIconified (WindowEvent e) ;
   public void windowDeiconified (WindowEvent e) ;
   public void windowActivated (WindowEvent e) ;
   public void windowDeactivated (WindowEvent e) ;
}
```

A WindowListener Class

```
/** This class handles Window Close events by prompting for confirmation
 *  that closing the window means shut down the application.
 */
public class MyWindowListener implements WindowListener {

  public void windowClosing (WindowEvent e) {

    int result = JOptionPane.showConfirmDialog // verify intent before exiting
      (e.getWindow(),                          // source of event
      "Are you sure you want to exit  ?",      // display message
      "Confirm Exit",                          // Title bar text
      JOptionPane.YES_NO_OPTION,               // button choices
      JOptionPane.QUESTION_MESSAGE);           // prompt icon

    if (result == JOptionPane.YES_OPTION) {
      System.exit(0);
    }
    return;    // we get here if "No" was chosen
  }

  // must provide ALL the methods of a "WindowListener", even if we're
  // not going to use the others...
  public void windowClosed (WindowEvent e) { }      // empty bodies
  public void windowOpened (WindowEvent e) { }
  public void windowIconified (WindowEvent e) { }
  public void windowDeiconified (WindowEvent e) { }
  public void windowActivated (WindowEvent e) { }
  public void windowDeactivated (WindowEvent e) { }
}
```

Using The Window Listener

```
/** This class is a JFrame with a Button which closes the Frame and immediately
 *  terminates the program; the Frame also allows Window Close to terminate after
 *  confirmation.
 */
public class WindowListenerFrame extends JFrame implements ActionListener {
  public WindowListenerFrame () {
    // ...code here to initialize the frame as usual...
    //create a button and add to frame
    JButton doneButton = new JButton ("Push Me To Quit Immediately");
    this.add(doneButton);
    // register this object (the frame) with the button as a listener
    doneButton.addActionListener(this);

    // create a separate "window listener" object
    MyWindowListener myWinListener = new MyWindowListener();

    // register the WindowListener object as a listener for window events
    // from this object (the frame).
    this.addWindowListener (myWinListener);

    // make sure the window doesn't hide when 'X' is clicked
    setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
    setVisible(true);
  }
  // action handler for the button (NOT for window events)
  public void actionPerformed (ActionEvent e) {
    System.exit(0);   //button push enters here, causes termination
  }
}
```

Run

Component Events

- **Generated when a Component is <u>moved</u>, <u>resized</u>, <u>shown</u>, or <u>hidden</u>**

- **Handled by “`ComponentListener`” objects**
  - Component listeners must implement the **`ComponentListener`** interface:

```
public interface ComponentListener {
    public void componentMoved (ComponentEvent e);
    public void componentResized (ComponentEvent e);
    public void componentHidden (ComponentEvent e);
    public void componentShown (ComponentEvent e);
}
```

Component Listeners

```
/**  This class handles Component events.  It provides only a componentResized()
 *    handler; other ComponentListener methods do nothing.
 */
public class MyComponentListener implements ComponentListener {
  private WindowResizeFrame myFrame ;
  public MyComponentListener (WindowResizeFrame theFrame) {
    myFrame = theFrame ;  // save a reference to the parent frame
  }
  public void componentResized (ComponentEvent e) {
    System.out.println("Frame size :“ +  myFrame.getWidth() +
                       " X " + myFrame.getHeight() );
  }
  // empty-body methods to complete ComponentListener implementation
  public void componentMoved (ComponentEvent e) {  } ;
  public void componentHidden (ComponentEvent e) {  } ;
  public void componentShown (ComponentEvent e) {  } ;
}
```

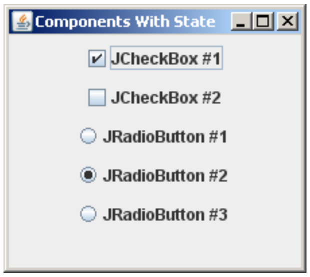
Using A ComponentListener

```
/** A JFrame which uses a Component Listener which handles window resize events.
 *  The frame passes a handle to itself to the ComponentListener so that the
 *  listener can obtain the new window size when the frame is resized.
 */
public class WindowResizeFrame extends JFrame implements ActionListener {
  public WindowResizeFrame () {
    // ...code here to initialize the frame to contain a button
    //    and make the frame the button listener...

    // attach a component listener to this frame
    MyComponentListener myCompListener = new MyComponentListener(this);
    this.addComponentListener (myCompListener);

    setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
    setVisible(true);
  }
  // action handler for the button -- terminates the program
  public void actionPerformed (ActionEvent e) {
    System.exit(0);
  }
}
```

Run

Item Events

- **Generated by certain components when their <u>state changes</u>**
  - **`JCheckBox`** **(** *<u>checked</u>* or *<u>unchecked</u>* **)**
  - **`JRadioButton`** **(** *<u>pushed or not pushed</u>* **)**
  - numerous other AWT/Swing components
- States: ***selected*** vs. ***not selected***



Components With State

☑ JCheckBox #1

□ JCheckBox #2

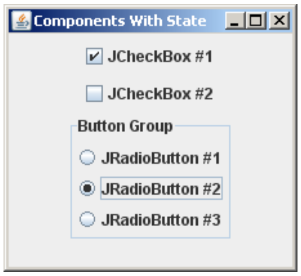
○ JRadioButton #1

● JRadioButton #2

○ JRadioButton #3

Item State Changes

- **Some components generate *multiple events***
  - E.g. **`ActionEvent`** ***and*** **`ItemEvent`**
    - **`ItemEvent`** generated due to *implicit* state change
      - Even when a component was not directly activated
    - Example: a radio button in a group



Run

Item Listeners

- **`ItemEvents` are handled by <u>Item Listener</u> objects**
- **Item Listeners must implement the `ItemListener` interface:**

```
public interface ItemListener {
        public void itemStateChanged (ItemEvent e);
}
```

Item State Change Example

```
/** A JFrame with a Checkbox that enables/disables a button and changes
 *  the text of a label.
 */
public class CheckboxButtonEnabler extends JFrame implements ItemListener {
  private JButton theButton ;
  private JCheckBox theCheckbox ;
  private JLabel theLabel ;
  public CheckboxButtonEnabler () {
    // code here to initialize the frame...
    // add some components to the frame
    theButton = new JButton ("Push Me");
    add(theButton);
    theLabel = new JLabel ("The button is enabled.");
    add(theLabel);
    theCheckbox = new JCheckBox ("Disable Button", false) ;
    add(theCheckbox) ;
    // register the frame as a listener for Item events from the checkbox
    theCheckbox.addItemListener(this);
    setDefaultCloseOperation (EXIT_ON_CLOSE);
    setVisible(true);
  }
  // ...continued
```

Item State Change Example (cont.)

```
  // ...continuation
  // this listener method is called when an "item event" has occurred
  public void itemStateChanged (ItemEvent e) {
    // When we get here, it is because the checkbox state has been changed;
    // determine the current checkbox state and set button and label accordingly
    if (theCheckbox.isSelected()) {
      // the checkbox is 'checked'; disable button and record in the label
      theButton.setEnabled(false);
      theLabel.setText("The button is disabled");
    }
    else{
      // the checkbox is 'unchecked'; enable button and record in the label
      theButton.setEnabled(true);
      theLabel.setText("The button is enabled");
    }
  }
}
```

Run

<< This page intentionally left (almost) blank >>

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

X - Interactive Techniques

- **Java AWT Graphics Class (and object)**
- **Component Repainting**
- **PaintComponent**
- **Mouse Motion**
- **Graphics State Saving**
- **RubberBanding**
- **Object Selection**

CSc 133 Lecture Notes

X - Interactive Techniques

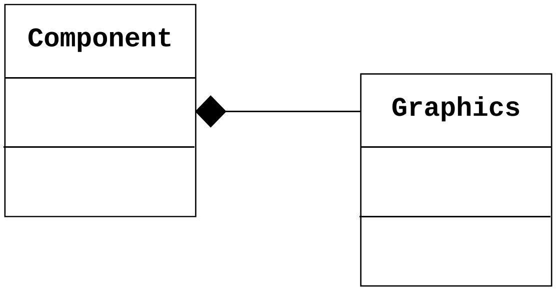
Computer Science Dept.
CSUS

Overview

- **Graphics Contexts**
- **Component Repainting**
- **Mouse Motion Events**
- **Graphics State Saving**
- **Onscreen Object Selection**

Component Graphics

- Every AWT `Component` contains an object of type `Graphics`



Component

Graphics

- `Graphics` objects know how to draw on the component

`Graphics` Class

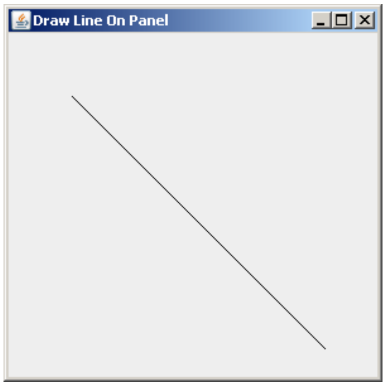
- `Graphics` objects contain methods to draw on their components
  - `drawLine (int x1, int y1, int x2, int y2);`
  - `drawRect (int x, int y,  int width, int height);`
  - `fillRect (int x, int y,  int width, int height);`
  - `drawOval (int x, int y,  int width, int height);`
  - `drawstring (String str,  int x,  int y);`
  - `setColor (Color c);`
  - `. . .`

JavaDoc

Using The Graphics Class

- You can get the graphics object by calling **`getGraphics()`** from the component you want to draw on.

```
myPanel = new JPanel();
myFrame.add(myPanel);
...
Graphics g = myPanel.getGraphics();
g.drawLine (50,50, 250,250);
```



Using Graphics

```
/** This class uses mouse events to draw graphical shapes on a panel. */
public class MouseGraphics extends JFrame implements MouseListener {
  private JPanel myPanel ;
  private Random myRNG = new Random();
  private final int FRAME_WIDTH=500, FRAME_HEIGHT=500;

  public MouseGraphics() {
    // ...code here to initialize the frame and add a JPanel which
    // ... has “this” (frame) as a MouseListener...
  }
  public void mousePressed (MouseEvent e) {
    // Draws a narrow (5x20) filled rectangle at the mouse location.
    Graphics g = myPanel.getGraphics();
    g.fillRect (e.getX(), e.getY(), 5, 20);
  }
  public void mouseClicked (MouseEvent e) {
    // Draws a small circle at a random location.
    int x = myRNG.nextInt(FRAME_WIDTH);
    int y = myRNG.nextInt(FRAME_HEIGHT);

    // draw radius 5 circle at x,y on the panel
    Graphics g = myPanel.getGraphics();
    g.drawOval (x,y, 10,10);
  }
  // ...code for additional MouseListener interface methods here...
}
```

Run

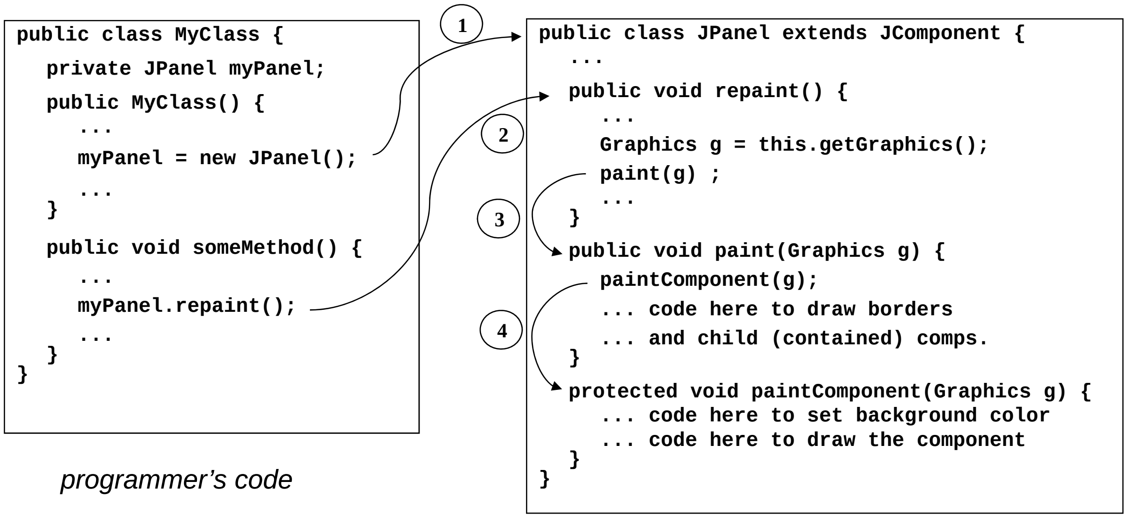
Component Repainting

- **Every AWT component has a `repaint()` method**
  - tells a component to update its screen appearance
  - Called by the OS window manager whenever the component needs redrawing
    - (Window restored, uncovered, ...)
  - Can also be called by the application code to force a redraw
    - **`setBackground(Color c)`** calls **`repaint()`**

Swing Components

- **Swing components also contain a method named <u>`paintComponent()`</u>**
  - **`repaint()`** passes the **`Graphics`** object to the component's **`paintComponent()`** method
  - **`paintComponent()`** is responsible for the actual drawing (using **`Graphics`**)
  - Never invoke **`paintComponent()`** directly; always call **`repaint()`**

paintComponent() Sequence



```
public class MyClass {
   private JPanel myPanel;
   public MyClass() {
      ...
      myPanel = new JPanel();
      ...
   }
   public void someMethod() {
      ...
      myPanel.repaint();
      ...
   }
}
```

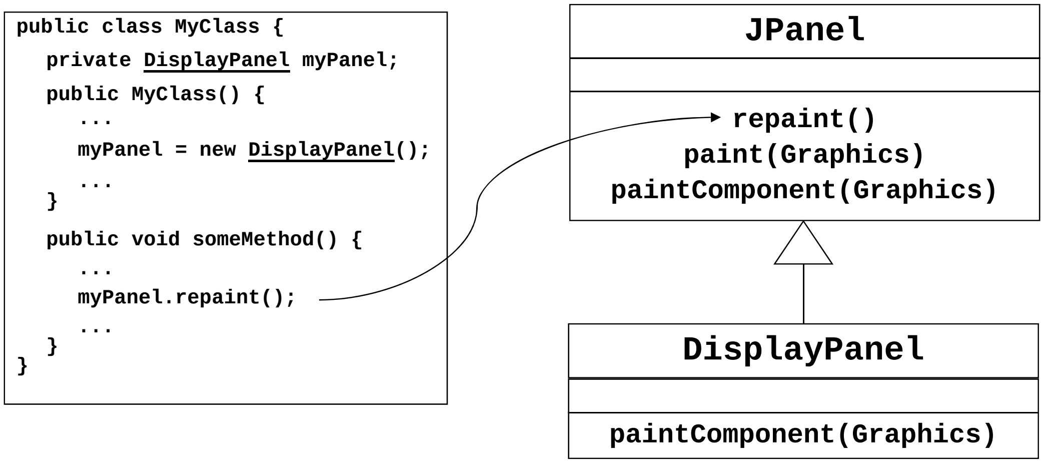
programmer's code

```
public class JPanel extends JComponent {
   ...
   public void repaint() {
      ...
      Graphics g = this.getGraphics();
      paint(g) ;
      ...
   }
   public void paint(Graphics g) {
      paintComponent(g);
      ... code here to draw borders
      ... and child (contained) comps.
   }
   protected void paintComponent(Graphics g) {
      ... code here to set background color
      ... code here to draw the component
   }
}
```

built in to Java

Overriding paintComponent()

- Consider the following organization
  - Which **paintComponent()** get invoked?



```
public class MyClass {
   private DisplayPanel myPanel;
   public MyClass() {
      ...
      myPanel = new DisplayPanel();
      ...
   }
   public void someMethod() {
      ...
      myPanel.repaint();
      ...
   }
}
```

Example 1

```
public class BoxDrawerFrame extends JFrame implements MouseListener {
  DisplayPanel myPanel ;
  public static Point mouseLoc = new Point(0,0);
  public BoxDrawerFrame() {
    setTitle ("BoxDrawer");
    setSize (400,400);
    myPanel = new DisplayPanel(this);
    myPanel.addMouseListener (this);
    myPanel.setSize (300,300);
    this.add (myPanel);
    this.setVisible (true);
  }
  public void mouseClicked (MouseEvent e) {
    // save current mouse position
    mouseLoc = e.getPoint();
    // force panel to be redrawn
    myPanel.repaint();
  }
  // ... code for additional MouseListener interface methods here...
}
```

Example 1 (cont.)

```
public class DisplayPanel extends JPanel {
  BoxDrawerFrame myFrame ;
  private int i = 1 ;
  public DisplayPanel (BoxDrawerFrame theFrame) {
    myFrame = theFrame ;
  }
  public void paintComponent (Graphics g) {
    super.paintComponent(g);
    g.drawString ("Hello " + i++, 200,200);
    g.drawRect(myFrame.mouseLoc.x, myFrame.mouseLoc.y, 20, 30);
  }
}
```

Run

Example 2

```
public class BoxSizerFrame extends JFrame implements MouseListener {
  DisplayPanel myPanel ;
  public static Point mouseStartLoc ;
  public static Point mouseEndLoc ;
  public BoxSizerFrame() {
    // code here to initialize frame as before...
    mouseStartLoc = new Point(0,0);
    mouseEndLoc = new Point (0,0);
    this.setVisible (true);
  }
  public void mousePressed (MouseEvent e) {
    mouseStartLoc = e.getPoint();          // save mouse start position
  }
  public void mouseReleased (MouseEvent e) {
    mouseEndLoc = e.getPoint();            // get new mouse position
    myPanel.repaint();                     // force panel to be drawn
  }
  // ... code here for additional MouseListener methods...
}
```

Example 2 (cont.)

```
public class DisplayPanel extends JPanel {
  BoxSizerFrame myFrame ;
  private int i = 1 ;
  public DisplayPanel (BoxSizerFrame theFrame) {
    myFrame = theFrame ;
  }
  public void paintComponent (Graphics g) {
    super.paintComponent(g);
    g.drawString ("Hello " + i++, 200,200);
    int width = myFrame.mouseEndLoc.x - myFrame.mouseStartLoc.x ;
    int height = myFrame.mouseEndLoc.y - myFrame.mouseStartLoc.y ;
    g.setColor (Color.red);
    g.drawRect(myFrame.mouseStartLoc.x, myFrame.mouseStartLoc.y,width,height);
  }
}
```

Run

Mouse Motion

- Moving a mouse generates an Event
- Handled by `MouseMotionListener` objects

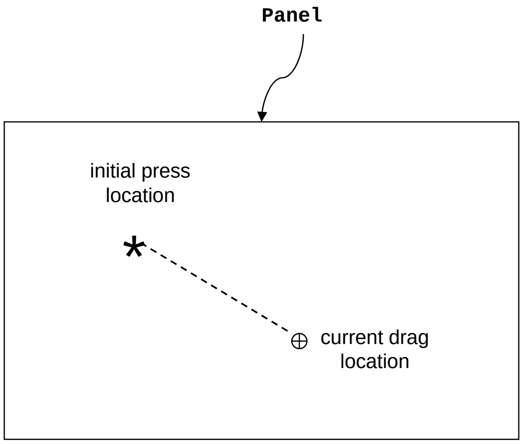
```
public interface MouseMotionListener {
   public void mouseDragged (MouseEvent  e);
   public void mouseMoved (MouseEvent  e);
}
```

- Drag vs. Move
  - Button down vs. no buttons down
- The events are called “MouseEvent”
  - no such thing as a “MouseMotionEvent”

Handling Mouse Motion

- Consider a Panel with the following listeners:

```
mousePressed()
{
 pressLoc = current mouse location;
}

mouseDragged()
{
 currentLoc = current mouse location;
 panel.repaint();
}

paintComponent (Graphics g)
{
 super.paintComponent(g);
 g.drawLine (pressLoc, currentLoc);
}
```



Handling Mouse Motion (cont.)

```
public class DisplayPanel extends JPanel
                        implements MouseListener, MouseMotionListener {
  private Point startPoint = null;         // mouse start location
  private Point currentPoint = null;       // mouse current location

  public DisplayPanel () {
    this.addMouseListener (this);
    this.addMouseMotionListener (this);
  }

  public void mousePressed (MouseEvent event) {
    startPoint = event.getPoint();         //save mouse pressed location
  }

  public void mouseDragged (MouseEvent event) {
    currentPoint = event.getPoint();       // save current drag location
    this.repaint();                        // force panel to be redrawn
  }

  public void paintComponent (Graphics g) {
    super.paintComponent(g);
    g.setColor (Color.green);
    if (startPoint != null && currentPoint != null) {
      g.drawLine (startPoint.x, startPoint.y, currentPoint.x, currentPoint.y);
    }
  }
  //...code for unused listener methods here...
}
```

Run

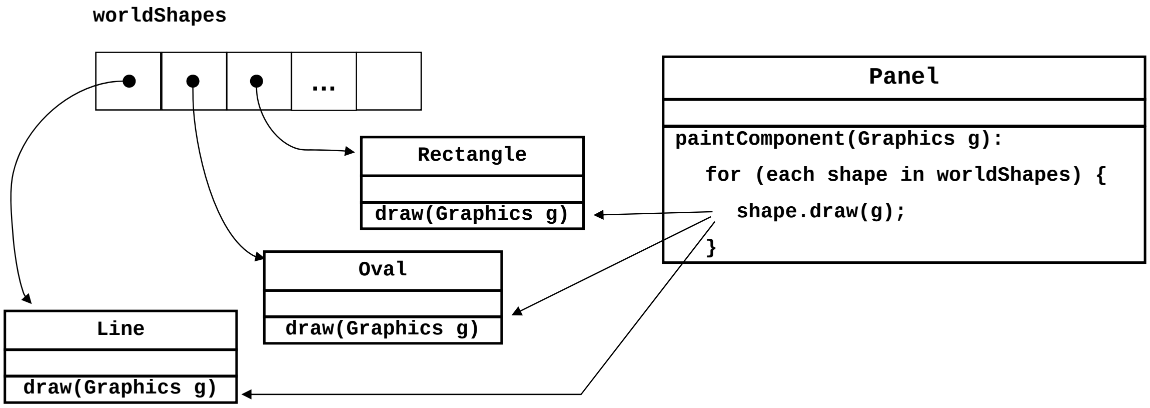
Mouse Handling Variation

```
public class DisplayPanel extends JPanel
                        implements MouseListener, MouseMotionListener {
  private Point prevPoint = null;          // location where last line ended
  private Point currentPoint = null;       // current mouse dragged location

  public DisplayPanel () {
    this.addMouseListener (this);
    this.addMouseMotionListener (this);
  }

  public void mousePressed (MouseEvent event) {
    prevPoint = event.getPoint();          //initial mouse pressed location
  }

  public void mouseDragged (MouseEvent event) {
    currentPoint = event.getPoint();       // get current mouse position
    Graphics g = this.getGraphics();       // get graphics context for panel
    g.setColor (Color.green);
    g.drawLine (prevPoint.x, prevPoint.y, currentPoint.x, currentPoint.y);
    prevPoint = currentPoint ;             // save ending point of draw
  }
  // ... Code here for unused listener methods...
}
```

Run

Maintaining Graphical State

- Must assume ***repaint()*** will be invoked
  - Must *keep track of objects you want displayed*
  - Redisplay them in paintComponent().



worldShapes

...

Panel

```
paintComponent(Graphics g):
   for (each shape in worldShapes) {
      shape.draw(g);
   }
```

Rectangle

draw(Graphics g)

Oval

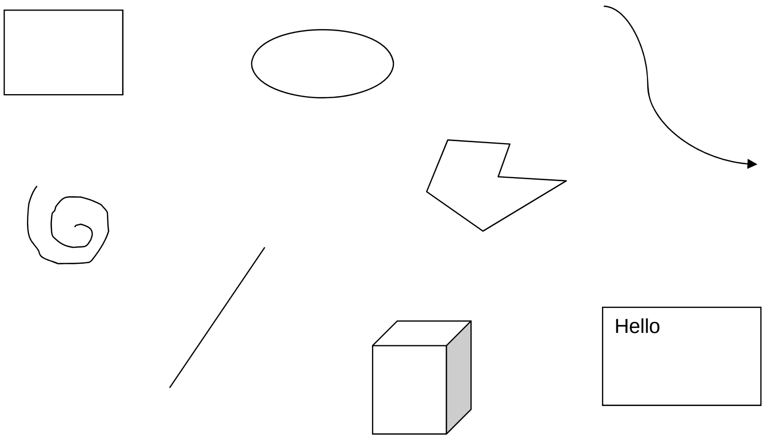
draw(Graphics g)

Line

draw(Graphics g)

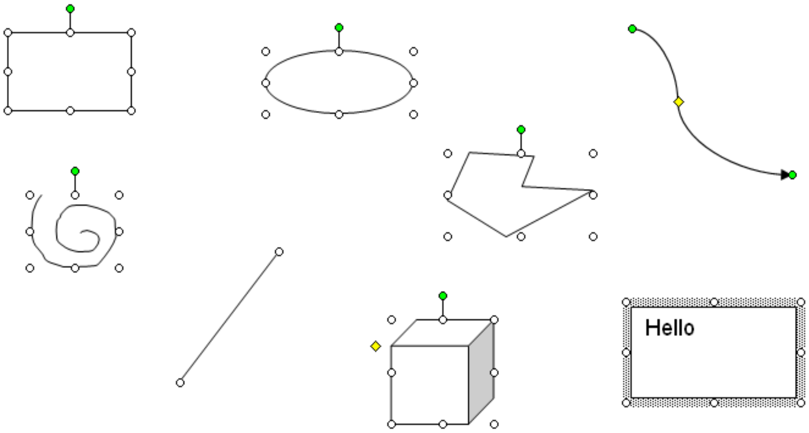
Object Selection

- Various *unselected* objects:



Object Selection (cont.)

- *Selected* versions of the same objects:

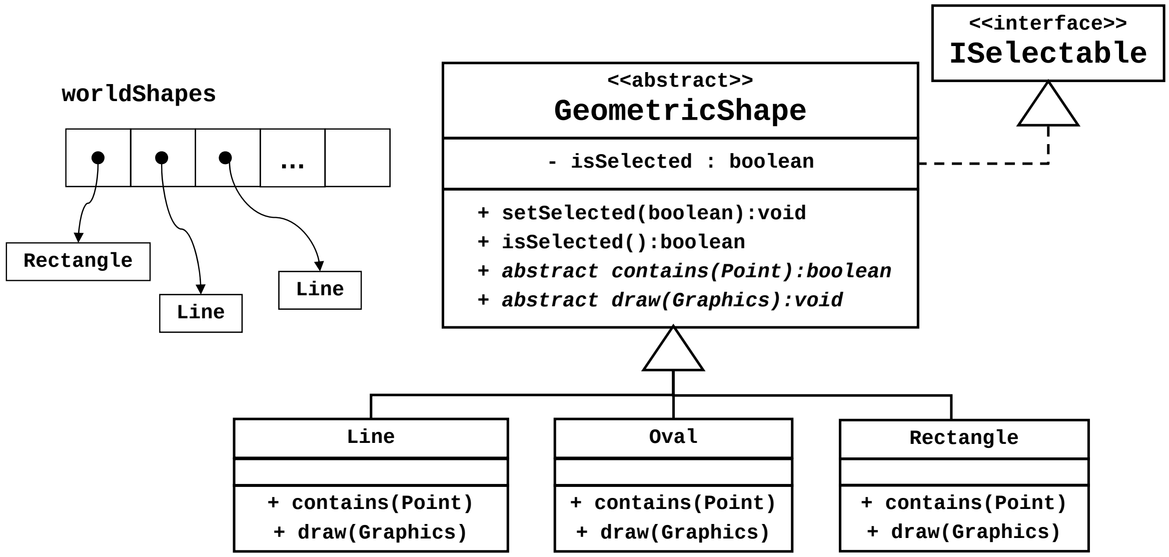


Defining “Selectability”

```
/** This interface defines the services (methods) provided
 *  by an object which is “Selectable” on the screen
 */
public interface ISelectable {

        // a way to mark an object as “selected” or not
        public void setSelected(boolean yesNo);

        // a way to test whether an object is selected
        public boolean isSelected();

        // a way to determine if a mouse point is “in” an object
        public boolean contains(Point p);

        // a way to “draw” the object that knows about drawing
        // different ways depending on “isSelected”
        public void draw(Graphics g);
}
```

Implementing Object Selection

(1) Expand objects to support selection



Implementing Object Selection (cont.)

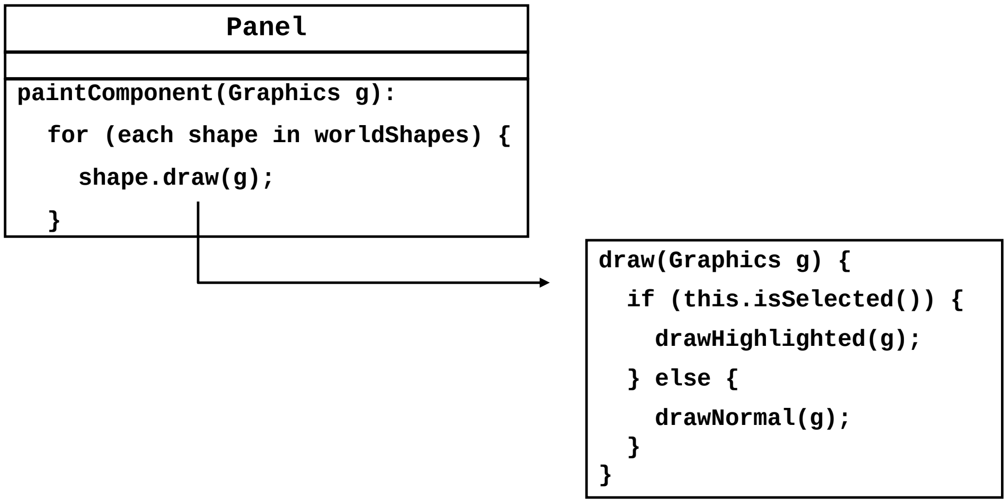
(2) On mousePressed:

- Determine if mouse is "inside" any shape
  - if shape contains mouse, mark as "selected"
- Repaint panel

```
void mousePressed(MouseEvent e) {
  Point p = e.getPoint();
  for (each shape in worldShapes) {
    if (shape.contains(p)) {
      shape.setSelected(true);
    } else {
      shape.setSelected(false);
    }
  }
  panel.repaint();
}
```

Implementing Object Selection (cont.)

(3) Draw “selected” objects in different form



```
Panel

paintComponent(Graphics g):
  for (each shape in worldShapes) {
    shape.draw(g);
  }
```

```
draw(Graphics g) {
  if (this.isSelected()) {
    drawHighlighted(g);
  } else {
    drawNormal(g);
  }
}
```

Object Selection Example

```
abstract public class GeometricShape implements ISelectable {
  private boolean isSelected;
  public void setSelected(boolean yesNo) { isSelected = yesNo; }
  public boolean isSelected() { return isSelected; }
  abstract void draw(Graphics g);
  abstract boolean contains(Point p);
}
```

```
public class Rectangle extends GeometricShape {
  private int width, height;
  ...
  public boolean contains(Point p) {
    int px = (int) p.getX();      // mouse location
    int py = (int) p.getY();
    int xLoc = getX();            // shape location
    int yLoc = getY();
    if ( (px >= xLoc) && (px <= xLoc+width)
      && (py >= yLoc) && (py <= yLoc+height) )
      return true; else return false;
  }
  public void draw(Graphics g) {
    if(isSelected())
      g.fillRect(getX(),getY(),width,height);
    else
      g.drawRect(getX(),getY(),width,height);
  }
}
```

```
public class ObjectSelectionFrame extends JFrame implements MouseListener {
  private Vector<GeometricShape> worldShapes = new Vector<GeometricShape>();
  public ObjectSelectionFrame() {
    // ...code here to initialize the frame with a DisplayPanel...
    worldShapes.addElement(new Rectangle(10,10,100, 50, Color.black));
    worldShapes.addElement(new Rectangle(100,300, 50,100, Color.red));
    ...
    this.setVisible (true);
  }
  public void mousePressed(MouseEvent e) {
    Point p = e.getPoint();
    for(int i=0;i<worldShapes.size();i++) {
      if(worldShapes.elementAt(i).contains(p))
        worldShapes.elementAt(i).setSelected(true);
      else
        worldShapes.elementAt(i).setSelected(false);
    }
    panel.repaint();
  }
  private class DisplayPanel extends JPanel {
    public void paintComponent(Graphics g) {
      super.paintComponent(g);
      for(int i=0; i<worldShapes.size();i++)
          worldShapes.elementAt(i).draw(g);
    }
  }
  //... definitions for other listener methods here ....
}
```

Run

XI – Introduction To Animation

- **Frame-based Animation**
- **Swing Timer Class**
- **Self-Drawing and Self-Animating Objects**
- **Computing Animated Location**
- **Collision Detection & Response**

CSc 133 Lecture Notes

XI - Introduction to Animation

Computer Science Dept.
CSUS

Overview

- Frame-based Animation
- Timers
- Moving Images
- Self-drawing and Self-animating Objects
- Collision Detection and Response

Frame-Based Animation

- Similar images shown in rapid succession imply movement

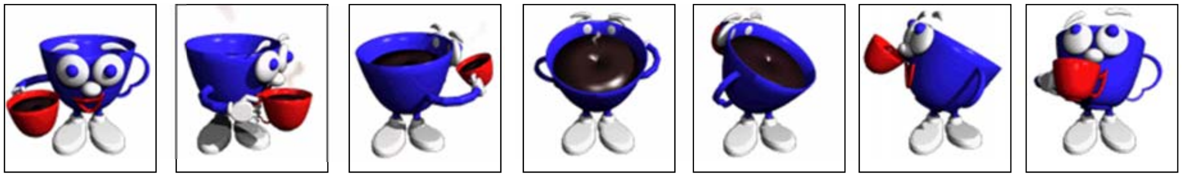
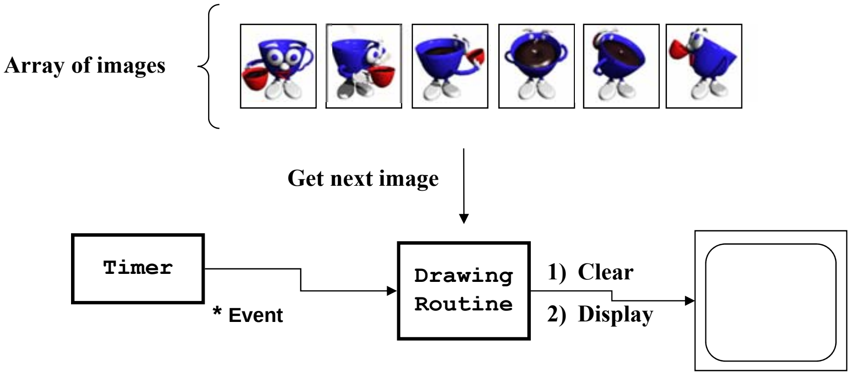


Image credit: Graphic Java: Mastering the JFC (3rd ed.), David Geary

Frame-Based Animation (cont.)

- Basic implementation structure:
  - Read images into an array
  - Use a Timer to invoke repeated “drawing”
  - Each “draw” outputs the “next” image



Array of images

Get next image

Timer

* Event

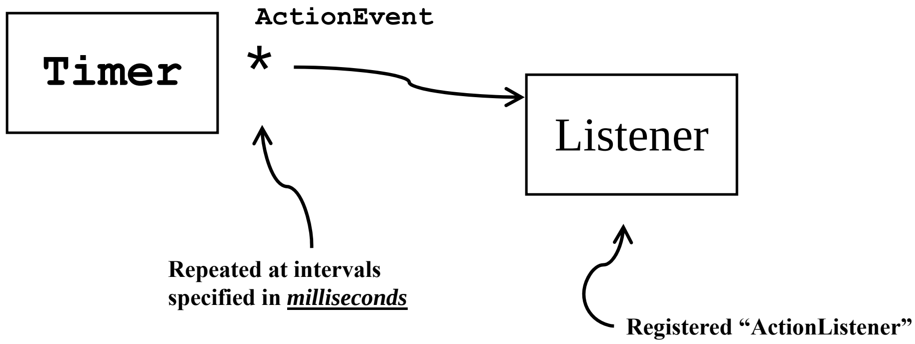
Drawing Routine

1) Clear

2) Display

Run

Java Swing Timer Class



Timer

ActionEvent

*

Listener

Repeated at intervals
specified in ***milliseconds***

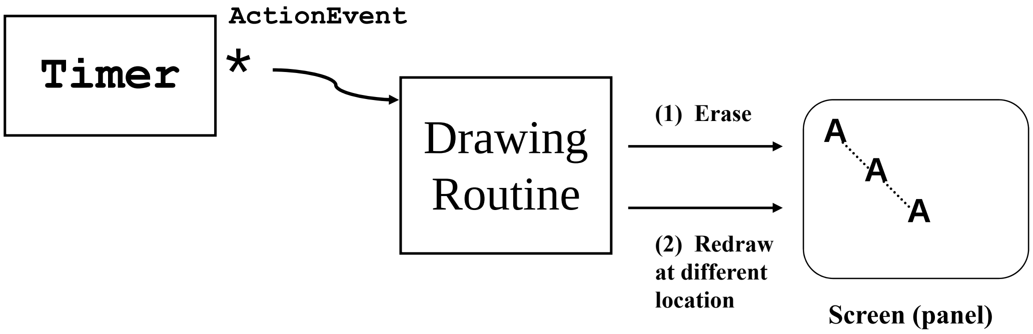
Registered “ActionListener”

Using the Swing Timer

```
/** This class creates and starts a Timer and provides a listener for ActionEvents from
 *   the Timer.  The listener draws graphical shapes of random sizes at random locations.
 */
public class TimerGraphics extends JFrame implements ActionListener {
  private JPanel myPanel ;
  private Random myRNG = new Random();
  private Timer myTimer = new Timer(1000,this) ;
  public TimerGraphics() {
    // ...code here to initialize the frame...
    // create a panel on which to do graphics; put it in the center
    myPanel = new JPanel();
    getContentPane().add (myPanel, "Center");
    // start the timer posting “interrupts” (ActionEvents)
    myTimer.start();
    ...
  }
  /** Entered when a Timer ActionEvent occurs; Draws an oval at a random location. */
  public void actionPerformed (ActionEvent e) {
    int xLoc = myRNG.nextInt(500);   // get a new random x,y location and size
    int yLoc = myRNG.nextInt(500);
    int xSize = myRNG.nextInt (50);
    int ySize = myRNG.nextInt (25);
    // draw a random-sized oval at a random location  on the panel
    Graphics g = myPanel.getGraphics();
    g.drawOval (xLoc,yLoc,xSize,ySize);
  }
}
```

Run

Animation via Image Movement



ActionEvent

Timer

Drawing Routine

(1) Erase

(2) Redraw at different location

Screen (panel)

The "drawing routine" for a JPanel is *<u>paint()</u>* + *<u>paintComponent()</u>*

Animation Example

```
/** A frame which starts a Timer which is used to repaint a display panel */
public class TimerAnimationFrame extends JFrame implements ActionListener
{
  private DisplayPanel myPanel ;
  private Timer timer ;
  private final int DELAY_IN_MSEC = 20 ;
  public TimerAnimationFrame() {
    // ...code here to initialize the frame with a panel as before...
    // create a Timer and start it firing ActionEvents which invoke this (frame)
    // as an ActionListener every "DELAY" milliseconds
    timer = new Timer (DELAY_IN_MSEC, this);
    timer.start() ;
    this.setVisible(true);
  }

  // handler for Timer ActionEvent: repaints the panel
  public void actionPerformed (ActionEvent e) {
    myPanel.repaint();
  }
}
```

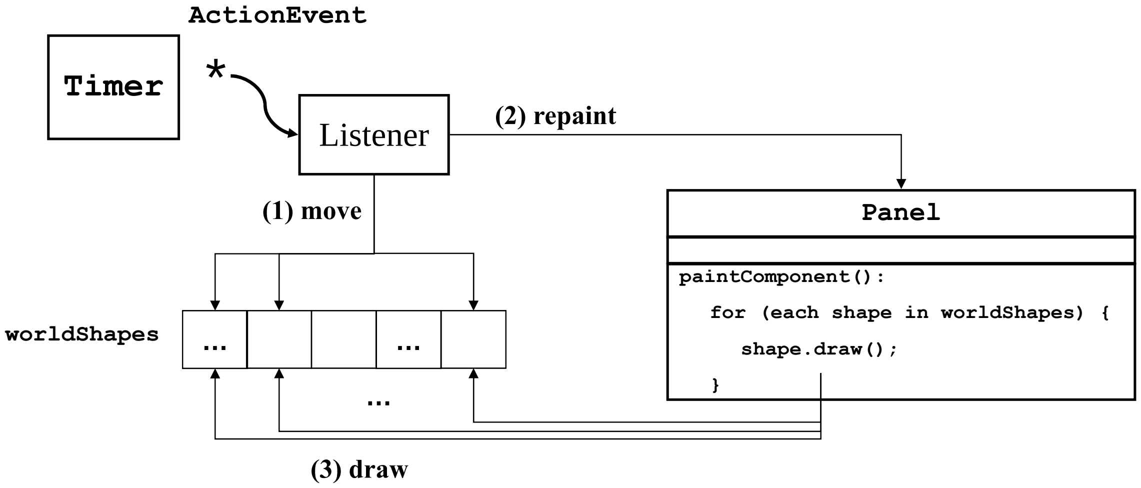
Animation Example (cont.)

```
public class DisplayPanel extends JPanel {
  private int currentX = 0, currentY = 0 ; // image location
  private int incX = 3, incY = 3 ;          // amount of movement
  private final int IMAGESIZE = 20 ;
  // update the image on the panel
  public void paintComponent (Graphics g) {
    super.paintComponent(g);
    // draw the image (a simple filled circle) at the current location.
    g.fillOval (currentX, currentY, IMAGESIZE, IMAGESIZE) ;
    // update the image position for the next draw
    currentX += incX ;
    currentY += incY ;
    // reverse the movement direction if the image reaches an edge
    if ( (currentX+IMAGESIZE >= this.getWidth()) || (currentX < 0) ) {
      incX = -incX ;
    }
    if ( (currentY+IMAGESIZE >= this.getHeight()) || (currentY < 0) ) {
      incY = -incY ;
    }
  }
}
```

Run

“Self-Animating” Objects

- Objects should be responsible for their own drawing and movement



```
paintComponent():
  for (each shape in worldShapes) {
    shape.draw();
  }
```

“Self-Animation” Example

```
/** A frame containing a collection of "self drawing objects". */
public class SelfDrawerAnimationFrame extends JFrame implements ActionListener {
  private DisplayPanel myPanel ;
  private Timer timer ;
  private Vector<WorldObject> theWorld ;
  public SelfDrawerAnimationFrame() {
    //...code here to initialize the frame with a BorderLayout
    theWorld = new Vector<WorldObject>(); // create a world containing a self-drawing object
    theWorld.add( new WorldObject() );
    //create a panel on which the world will be drawn
    myPanel = new DisplayPanel(theWorld) ;
    this.add (myPanel, BorderLayout.CENTER);
    // create a Timer and start it firing ActionEvents
    timer = new Timer (DELAY_IN_MSEC, this);
    timer.start() ;
    this.setVisible(true);
  }
  // handler for Timer ActionEvent: tells object to move itself, then repaints the panel
  public void actionPerformed (ActionEvent e) {
    int xMax = myPanel.getWidth(), yMax = myPanel.getHeight() ;
    for (WorldObject obj : theWorld) {
      obj.move (xMax, yMax);
    }
    myPanel.repaint();
  }
}
```

“Self-Animation” Example (cont.)

```
/** This class defines an object which knows how to "move" itself, given a panel
 *  with boundaries, and knows how to "draw“ itself given a Graphics object.
*/
public class WorldObject {
  private int currentX = 0, currentY = 0 ; // the object's current location
  private int incX = 3, incY = 3 ;         // amount of movement on each move
  private final int IMAGESIZE = 35 ;       // object-specific image size
  // create the image to be used for this object
  private Image theImage = Toolkit.getDefaultToolkit().getImage("happyFace.gif");
  // move this object within the specified boundaries
  public void move (int xMax, int yMax) {
    // update the object position
    currentX += incX ;
    currentY += incY ;
    // reverse the next movement direction if the location has reached an edge
    if ( (currentX+IMAGESIZE >= xMax) || (currentX < 0) ) {
      incX = -incX ;
    }
    if ( (currentY+IMAGESIZE >= yMax) || (currentY < 0) ) {
      incY = -incY ;
    }
  }
  // draw the representation of this object using the received Graphics context
  public void draw(Graphics g) {
    g.drawImage (theImage, currentX, currentY, null) ;
  }
}
```

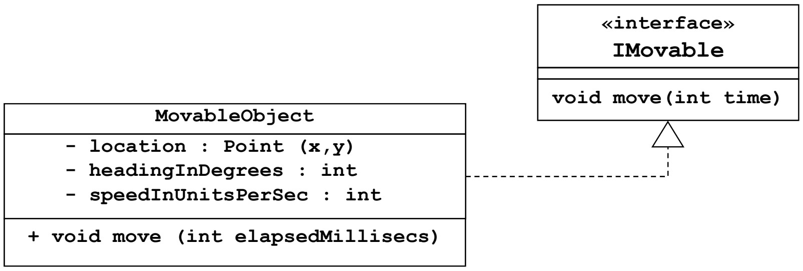
“Self-Animation” Example (cont.)

```
/** A panel which which redraws its world object(s) each time
 *  the panel is repainted.
 */
public class DisplayPanel extends JPanel {

  Vector<WorldObject> theWorld ;

  public DisplayPanel (Vector<WorldObject> world) {
    theWorld = world ;
  }

  public void paintComponent (Graphics g) {
    super.paintComponent(g);
    for (WorldObject obj : theWorld) {
      obj.draw(g)  ;
    }
  }
}
```

Run

Computing Animated Location

- Consider a “moveable object” defined as:



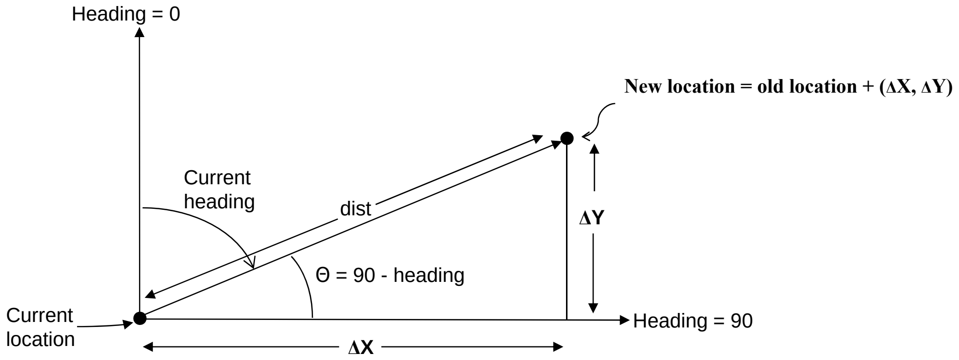
```
«interface»
IMovable
void move(int time)
```

```
MovableObject
  - location : Point (x,y)
  - headingInDegrees : int
  - speedInUnitsPerSec : int
+ void move (int elapsedMillisecs)
```

- Calling **move()** instructs the object to update its location, determined by
  - How long it has been moving from its current location
  - Its current heading and speed

Computed Animated Location (cont.)

Computing a new location:



$$dist = rate \times time \;=\; \text{speedInUnitsPerSecond} \times \frac{\text{elapsedMilliSecs}}{1000}$$

$$\cos\theta = \frac{\Delta X}{dist}; \text{ so } \Delta X = \cos\theta \times dist. \text{ Likewise, } \Delta Y = \sin\theta \times dist$$

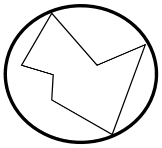
Collision Detection

- **Moving objects require:**
  - ***Detecting collisions***
  - ***Dealing with (responding to) collisions***
- ***Detection == determining overlap***
  - ***Complicated by “shape”***

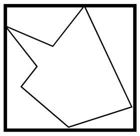
Collision Detection (cont.)

Simplification: “bounding volumes”

- ***Areas*** **in the 2D case**



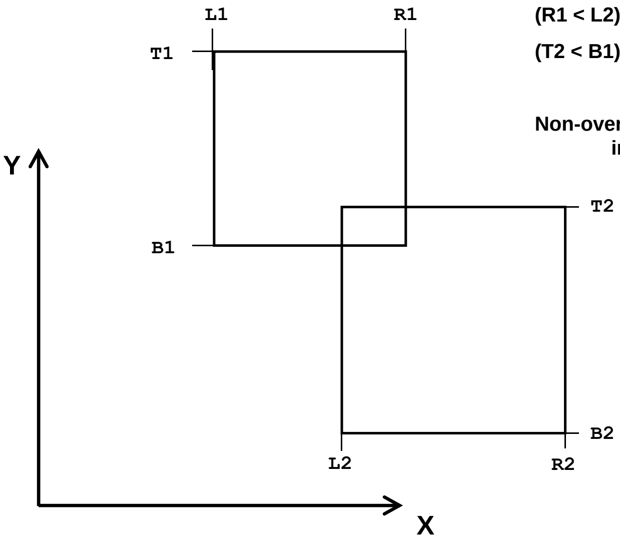
Bounding Circle



Bounding Rectangle

Collision Detection (cont.)

Bounding rectangle collisions



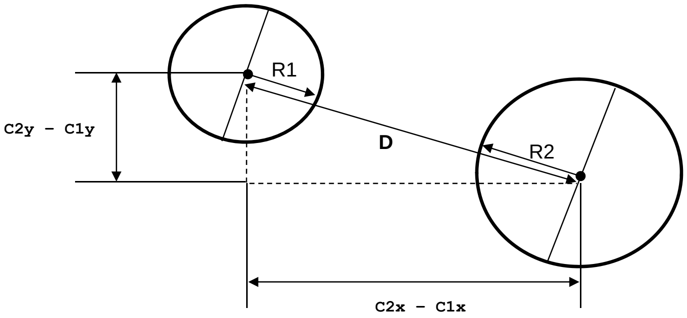
(R1 < L2) OR (L1 > R2) → No Left/Right overlap

(T2 < B1) OR (T1 < B2) → No Top/Bottom overlap

Non-overlap in *__either__* *(i.e., one or the other or BOTH)* implies *no collision*

Collision Detection (cont.)

Bounding circle collisions



$D^2 = (C2y - C1y)^2 + (C2x - C1x)^2$

$D \leq (R1+R2)$ ➔ colliding (requires calculating sqrt)

Also, $D^2 \leq (R1+R2)^2$ ➔ colliding (no sqrt)

Collision Response

- Application-dependent
  - Modify heading
  - Change appearance
  - Delete (explode?)
  - Update application state (e.g. “score points”)
  - Other …

Collision Response (cont.)

- `Collider` interface

```
public interface ICollider {
  public boolean collidesWith(ICollider otherObject);
  public void handleCollision(ICollider otherObject);
}
```

  - `collidesWith()`: apply appropriate *detection* algorithm
  - `handleCollision()`: apply appropriate *response* algorithm

Collider Example

```
/** A frame with self drawing objects. A Timer instructs the objects to move and
 *  a panel to redraw the objects. On collision, an object changes color.  */
public class CollisionFrame extends JFrame implements ActionListener {
  private DisplayPanel myPanel ;
  private Timer timer ;
  private Vector theWorld ;
  public CollisionFrame() {
    // code here to initialize the frame...
    // create a world containing objects
    theWorld = new Vector();
    Dimension worldSize = new Dimension(500, 400);
    addObjects(worldSize);
    // create a panel on which the world objects will be drawn
    myPanel = new DisplayPanel(theWorld) ;
    this.getContentPane().add (myPanel, BorderLayout.CENTER);
    // create a Timer to invoke move and repaint operations
    timer = new Timer (20, this); //20msec delay
    timer.start() ;
    this.setVisible(true);
  }
  private void addObjects(Dimension worldSize) {
    theWorld.addElement(new WorldObject(Color.red, worldSize));
    theWorld.addElement(new WorldObject(Color.blue, worldSize));
    // ...code here to add additional world objects...
  }
...continued...
```

Collider Example (cont.)

```
// this method is entered on each Timer tick; it moves the objects, checks for collisions
// and invokes the collision handler, then repaints the display panel.
public void actionPerformed (ActionEvent e) {
  // use the panel to define move/rebound limits
  Dimension limits = new Dimension (myPanel.getWidth(), myPanel.getHeight());
  // move all the world objects
  Iterator iter = theWorld.getIterator();
  while (iter.hasNext()) {
    ( (IMovable) iter.next()).move(limits);
  }
  // check if moving caused any collisions
  iter = theWorld.getIterator();
  while (iter.hasNext()) {
    ICollider curObj = (ICollider) iter.next(); // get a collidable object
    // check if this object collides with any OTHER object
    Iterator iter2 = theWorld.getIterator();
    while (iter2.hasNext()) {
      ICollider otherObj = (ICollider) iter2.next(); // get a collidable object
        if (otherObj != curObj) {        // make sure it's not the SAME object
        // check for collision
        if (curObj.collidesWith(otherObj)) {
          curObj.handleCollision(otherObj);
        }
      }
    }
  }
  myPanel.repaint(); // redraw the world
 }
} //end class CollisionFrame
```

Collider Example (cont.)

```
/** This class defines an object which knows how to "move" and “draw” itself, and
 *  how to determine whether it collides with another object, and provides a method
 *  specifying what to if it is instructed to handle a collision with another object.
 *  (In this case collision changes the color of the object.)  */
public class WorldObject implements IMovable, IDrawable, ICollider {
  private static Random worldRNG = new Random();    // random number generator
  public void move (Dimension limits) { ... }       // as before
  public void draw(Graphics g) { ... }              // as before
  // Use bounding circles to determine whether this object has collided with another
  public boolean collidesWith(ICollider obj) {
    boolean result = false;
    int thisCenterX = this.xLoc + (OBJECT_SIZE/2); // find centers
    int thisCenterY = this.yLoc + (OBJECT_SIZE/2);
    int otherCenterX = obj.getX() + (OBJECT_SIZE/2);
    int otherCenterY = obj.getY() + (OBJECT_SIZE/2);
    // find dist between centers (use square, to avoid taking roots)
    int dx = thisCenterX - otherCenterX;
    int dy = thisCenterY - otherCenterY;
    int distBetweenCentersSqr = (dx*dx + dy*dy);
    // find square of sum of radii
    int thisRadius = OBJECT_SIZE/2;
    int otherRadius = OBJECT_SIZE/2;
    int radiiSqr = (thisRadius*thisRadius + 2*thisRadius*otherRadius
                                           + otherRadius*otherRadius);
    if (distBetweenCentersSqr <= radiiSqr) { result = true ; }
    return result ;
  }
```

Collider Example (cont.)

```
  ...
  // defines this object's response to a collision with otherObject
  public void handleCollision(ICollider otherObject) {
    // change my color by generating three random colors
    myColor = new Color ( worldRNG.nextInt(256),
                          worldRNG.nextInt(256),
                          worldRNG.nextInt(256) );
  }
  // ...additional required interface methods here...
} // end class WorldObject

------------------------------

/** A panel which which redraws its object(s) each time it is repainted. */
public class DisplayPanel extends JPanel {
  Collection theWorld ;
  public DisplayPanel (Collection aWorld) {
    theWorld = aWorld ;
  }
  public void paintComponent (Graphics g) {
    super.paintComponent(g);
    Iterator iter = theWorld.getDrawableIterator();
    while (iter.hasNext())        {
      IDrawable next = (WorldObject) iter.next();
      next.draw(g) ;
    }
  }
}
```

Run

<< This page intentionally left (almost) blank >>

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

XII – Introduction To Sound

- **Sampled Audio**
- **Sound file formats**
- **Sound APIs**
- **Java AudioClips**
- **Path and Platform-Independence**
- **Java Sound API**
- **OpenAL**

CSc 133 Lecture Notes

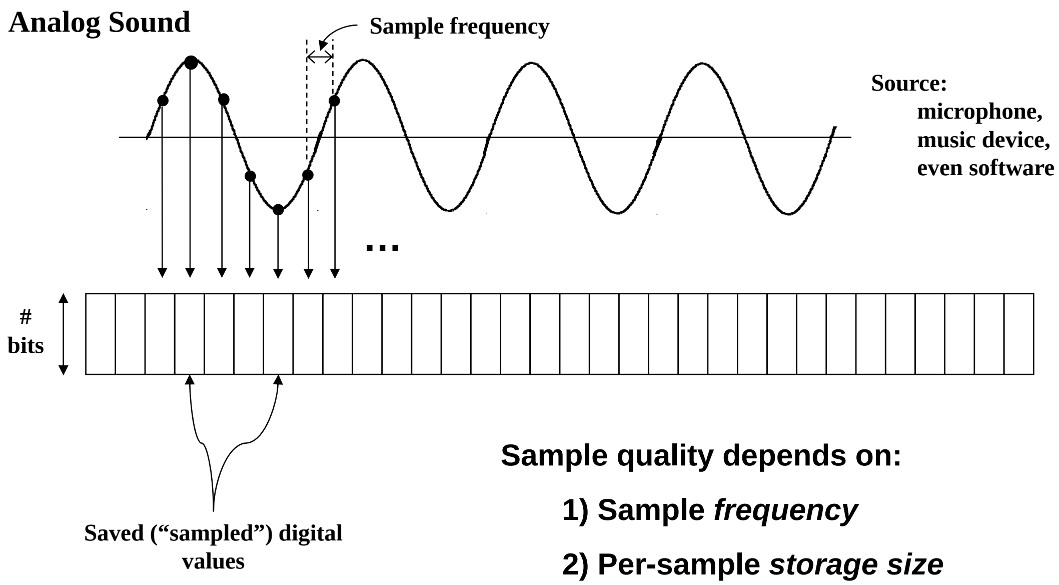
XII - Introduction to Sound

Computer Science Dept.
CSUS

Overview

- **Sound Files**
- **Sound APIs**
- **Using Java `AudioClip`**

Sampled Audio



Analog Sound

Sample frequency

Source:
microphone,
music device,
even software

...

bits

Saved ("sampled") digital values

Sample quality depends on:

1) Sample *frequency*

2) Per-sample *storage size*

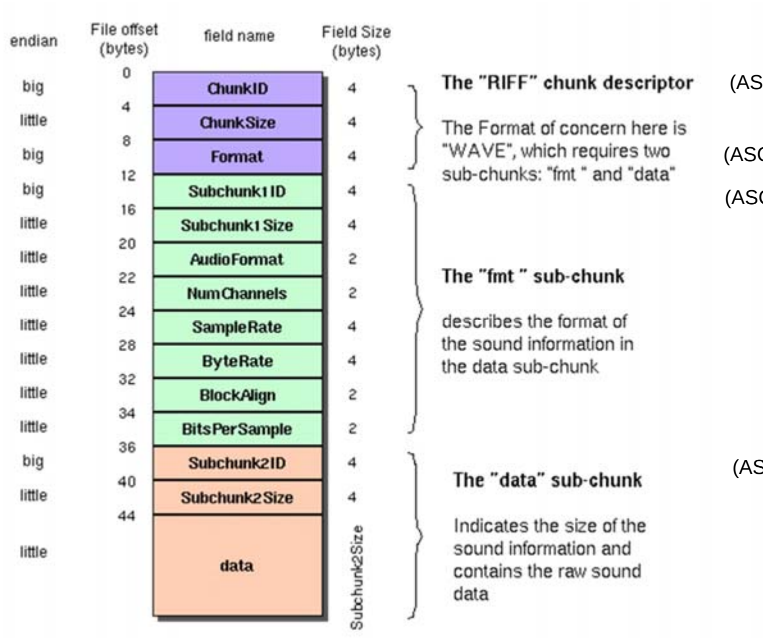
(bits per sample)

Sound File Formats

`.au`	Sun Audio File (Unix/Linux)
`.aiff`	Audio Interchange File Format (Mac)
`.cda`	CD Digital Audio (track information)
`.mpx`	MPEG Audio (mp, mp2, mp3, mp4)
`.mid`	MIDI file (sequenced, not sampled)
`.ogg`	Ogg-Vorbis file (open source)
`.ra`	Real Audio (designed for streaming)
`.wav`	Windows "wave file"

Finding sound files: **`www.findsounds.com`**

Example: WAVE Format



endian	File offset (bytes)	field name	Field Size (bytes)
big	0	ChunkID	4
little	4	ChunkSize	4
big	8	Format	4
big	12	Subchunk1ID	4
little	16	Subchunk1 Size	4
little	20	AudioFormat	2
little	22	NumChannels	2
little	24	SampleRate	4
little	28	ByteRate	4
little	32	BlockAlign	2
little	34	BitsPerSample	2
big	36	Subchunk2ID	4
little	40	Subchunk2Size	4
little	44	data	Subchunk2Size

The "RIFF" chunk descriptor

The Format of concern here is "WAVE", which requires two sub-chunks: "fmt " and "data"

The "fmt " sub-chunk

describes the format of the sound information in the data sub-chunk

The "data" sub-chunk

Indicates the size of the sound information and contains the raw sound data

(ASCII “RIFF”, 0x52494646)

(ASCII “WAVE”, 0x57415645)

(ASCII “fmt ”, 0x666D7420)

(ASCII “data”, 0x64617461)

Image credit: http://ccrma.stanford.edu/courses/422/projects/WaveFormat/

Popular Sound API’s

- Java **`AudioClip`** Interface
- JavaSound
- DirectSound / DirectSound3D
- Linux Open Sound System (OSS)
- Advanced Linux Sound Architecture (ALSA)
- OpenAL / JOAL

Java AudioClips

- Originally part of web-centric **`Applets`**
- Supports
  - Automatic loading
  - **`play(), loop(), stop()`**
    - No way to determine progress or completion
- Supported sound file types depend on JVM
    - Sun default JVM: *.wav*, *.aiff*, *.au* , *.mid,* others…

AudioClip Interface

```
public interface AudioClip {
  public void play();
  public void loop();
  public void stop();
}
```

Using Java AudioClips

```
import java.applet.Applet ;
import java.applet.AudioClip ;
import java.io.File ;
...
/** This method constructs a Java “AudioClip” object from the
 *  specified file, then plays the AudioClip.
 */
public void playSound (String fileName) {
   try {
      AudioClip  clip = Applet.newAudioClip(
                                   new File(fileName).toURI().toURL());
      clip.play();
   }
   catch (Exception e) {
      throw new RuntimeException("Problem with " + fileName + ": " + e);
   }
}
```

Encapsulating AudioClips

```
/** This class encapsulates a sound file as an AudioClip inside a
 *  “Sound” object, and provides a method for playing the Sound.
 */
public class Sound {
  AudioClip myClip ;
  public Sound(String fileName) {
    try {
      File file = new File(fileName);
      if (file.exists()) {
          myClip = Applet.newAudioClip(file.toURI().toURL());
      } else {
          throw new RuntimeException("Sound: file not found: " + fileName);
      }
    } catch (MalformedURLException e) {
        throw new RuntimeException("Sound: malformed URL: " + e);
    }
  }
  public void play() {
    myClip.play();
  }
}
```

Run

Path & Platform Independence

- Avoid hard-coding “locations”
  - Use “.” (current location-relative)
- Avoid OS-specific separators (“\” vs. “/”)
  - Use Java’s **`File.separator`** constant
- Example:

```
String soundDir = "." + File.separator + "sounds" + File.separator ;
String fileName = "dog.wav";
String filePath = soundDir + fileName ;
Sound bark = new Sound(filePath);
```

Using Encapsulated Sounds

```
/** This class defines a “world object” containing an encapsulated
 * AudioClip which is played when a collision with another object occurs.
 */
public class WorldObject implements IMovable, IDrawable, Icollider {
  private static Random worldRNG = new Random();
  private int xLoc, yLoc ;
  private Color myColor ;
  private Sound myCollisionSound ;
  ...

  public WorldObject (Color aColor, Dimension bounds) {
    myColor = aColor ;
    xLoc = worldRNG.nextInt((int)bounds.getWidth()-OBJECT_SIZE);
    yLoc = worldRNG.nextInt((int)bounds.getHeight()-OBJECT_SIZE);

    String soundDir = "." + File.separator + "sounds" + File.separator ;
    String collisionSoundFile = "dog.wav" ;
    String collisionSoundPath = soundDir + collisionSoundFile ;

    myCollisionSound = new Sound(collisionSoundPath) ;
  }

  //continued...
```

Using Encapsulated Sounds (cont.)

```
    //class WorldObject, continued...

    public void move (Dimension bounds) { ... }              //as before
    public void draw(Graphics g) { ... }                     //as before
    public boolean collidesWith(ICollider obj) { ... }       //as before
    public boolean contains (int x, int y) { ... }           //as before

    ...

    //define this object's response to a collision with otherObject
    public void handleCollision(ICollider otherObject) {
      //change my color
      myColor = new Color ( worldRNG.nextInt(256),
                            worldRNG.nextInt(256),
                            worldRNG.nextInt(256));
      //play my collision sound
      myCollisionSound.play();
    }

    ...

  }//end class WorldObject
```

Run

Java Sound API

- A *package* of expanded sound support

  import javax.sound.sampled;

  import javax.sound.midi;

- New capabilities:
  - Skip to a specified file location
  - Control volume, balance, tempo, track selection, etc.
  - Create and manipulate sound files
  - Support for streaming
- Some shortcomings
  - Doesn't recognize some common file characteristics
  - Doesn't support spatial ("3D") sound

openAL

- “Open Audio Library”
  - 3D Audio API (**`www.openal.org`**)
- Open-source
- Cross-platform
- Modeled after OpenGL
- Java binding (“JOAL”):

  `www.jogamp.org`

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

XIII – Transformations

- **Affine Transforms**
- **Translation**
- **Rotation**
- **Scaling**
- **Matrix Representation**
- **Homogeneous Coordinates**
- **Concatenation of Transforms**

CSc 133 Lecture Notes

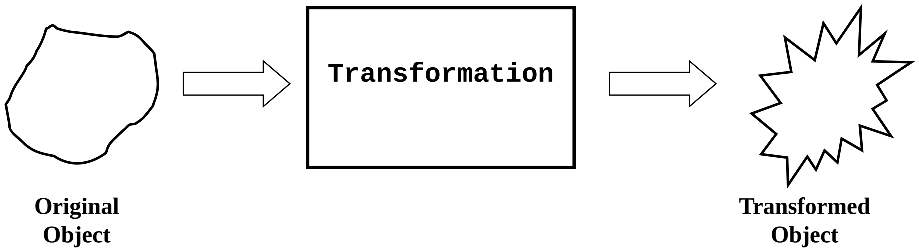
XIII - Transformations

Computer Science Dept.
CSUS

Overview

- Affine Transformations
- Transforming Points & Lines
- Matrix Representation of Transforms
- Homogeneous Coordinates
- Concatenation of Transformations

The “Transformation” Concept



Transformation

Original Object

Transformed Object

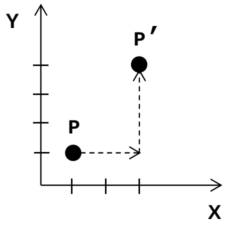
- “Original object” could be anything
    - We will focus on geometric objects
- “Transformed object” is usually (*but not necessarily*) of same type

“Affine” Transformations

- Properties:
    - “Map” (transform) finite points into finite points
    - Map parallel lines into parallel lines
- Common examples used in graphics:
    - Translation
    - Rotation
    - Scaling

Transformations on Points

- Translation



$P = (x,\ y)$

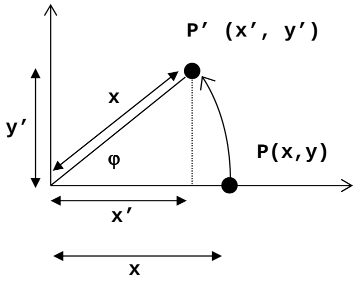
$T = (+2,\ +3)$

$P' = (x+2,\ \ y+3)$

$$P \rightarrow \boxed{T} \rightarrow P' \qquad \text{or} \qquad P' \leftarrow \boxed{T} \leftarrow P$$

Transformations on Points (cont.)

- Rotation <u>about the origin</u> (point on X axis)



$\cos(\varphi) = x' / x$; hence

$x' = x \cos(\varphi)$

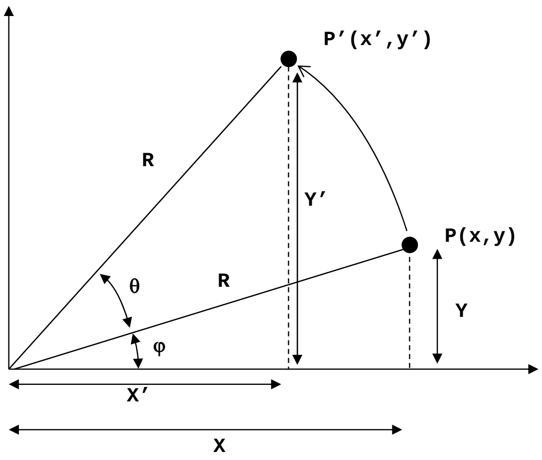
$\sin(\varphi) = y' / x$; hence

$y' = x \sin(\varphi)$

$$P \rightarrow \boxed{R} \rightarrow P' \qquad \text{or} \qquad P' \leftarrow \boxed{R} \leftarrow P$$

Transformations on Points (cont.)

- Rotation about the origin (arbitrary point)



$\cos(\varphi) = X / R$ and $\sin(\varphi) = Y / R;$

$X = R\cos(\varphi)$ and $Y = R\sin(\varphi)$

$X' = R\cos(\varphi + \theta)$

$= R\,(\cos(\varphi)\cos(\theta) - \sin(\varphi)\sin(\theta))$

$= \underline{R\cos(\varphi)}\,\cos(\theta) - \underline{R\sin(\varphi)}\,\sin(\theta)$

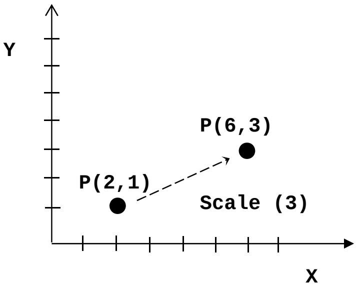
$= \underline{X}\,\cos(\theta) - \underline{Y}\,\sin(\theta)$

Similarly,

$Y' = X\sin(\theta) + Y\cos(\theta)$

Transformations on Points (cont.)

- Scaling
  - Multiplication by a “scale factor”



$P = (x, y)$

$S = (s_x, s_y)$

$P' = (x*s_x, y*s_y)$

$P \rightarrow \boxed{S} \rightarrow P'$ or $P' \leftarrow \boxed{S} \leftarrow P$

Transformations on Points (cont.)

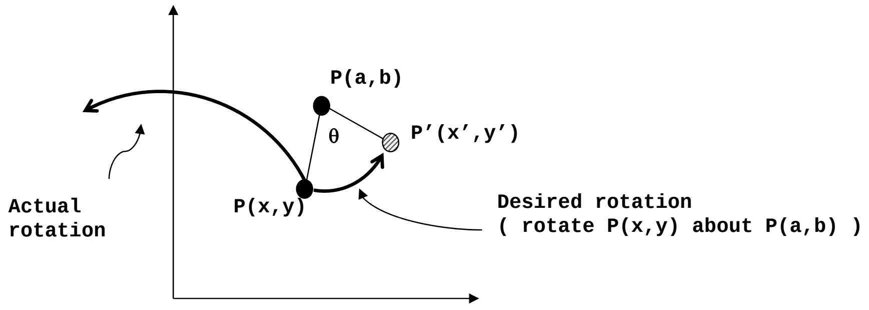
- Scaling is
  - Relative to the origin (like rotation)
  - *Different* from a “move”:
    - Translate (3,3) always moves exactly 3 units
    - Scale (3,3) depends on the initial point being scaled:

```
P(1,1)*Scale(3,3) →  P’(3,3)    (“move” of 2)
P(4,4)*Scale(3,3) →  P’(12,12)  (“move” of 8)
```

- Scaling by a fraction: move “closer to origin”
- Scaling by a negative value: “reflection” across axes (“mirroring”)
- Scaling where $s_x \neq s_y$: change “aspect ratio”

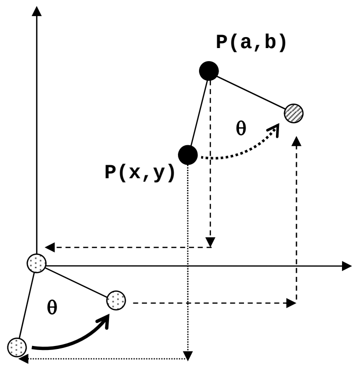
Transformations on Points (cont.)

- Rotating a point about an arbitrary point
  - Problem: rotation formulas are ***relative to the origin***



Transformations on Points (cont.)

- Solution:
  - Translate to origin
  - Perform rotation
  - Translate “back”

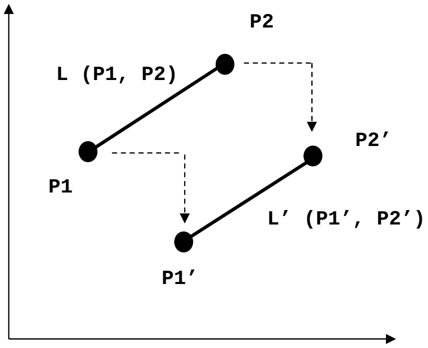


1. Translate P(x,y) by (-a, -b)
2. Rotate (translated) P
3. “Undo” the translation
   (translate result by (+a, +b))

[Note we are assuming the effect of a *sequence* of affine transforms is still affine (it is…)]

Transformations on Lines

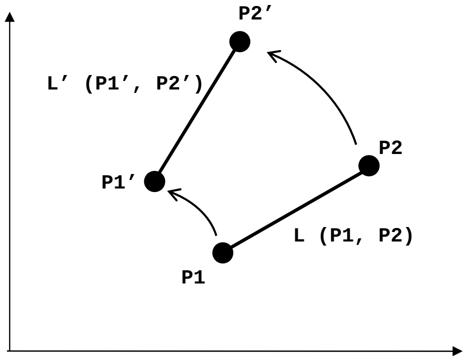
- Translation: translate the endpoints



- Translate (Line(p1,p2))
  = Line (Translate(p1), Translate(p2))
- A result of the property that translation is affine

Transformations on Lines (cont.)

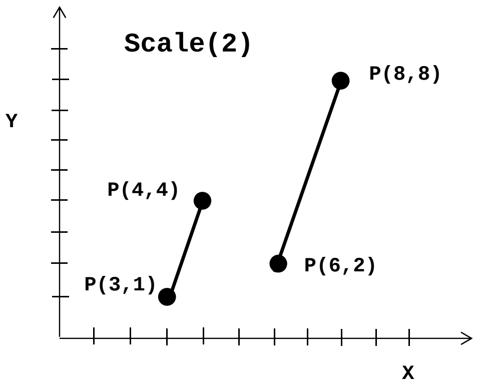
- Rotation *about the origin*: rotate the endpoints



- **Rotate (Line(p1,p2))**
  = Line (Rotate(p1), Rotate(p2))
- **A result of the property that rotation about the origin is affine**

Transformations on Lines (cont.)

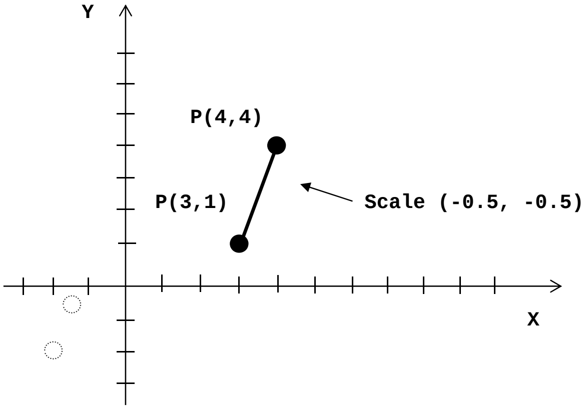
- Scaling: scale the endpoints



- **Scale (Line(p1,p2))**
  = Line (Scale(p1), Scale(p2))
- **A result of the property that scaling is affine**
- **Note how scale seems to “move” also**

Transformations on Lines (cont.)

- Question: what is the result of `Scale(-0.5, -0.5)` applied to this line?



Some general rules for __scaling__:

- Absolute Value of Scale Factor > 1 → "bigger"
- Absolute Value of Scale Factor < 1 → "smaller"
- Scale Factor < 0 → "flip" ("mirror")

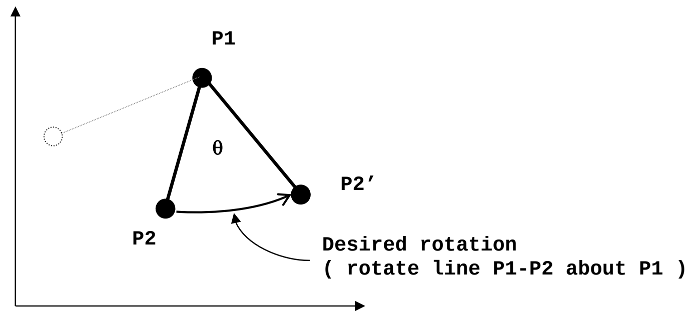
Identity Operations:

- For translation: 0 → No Change
- For rotation: 0 → No Change
- For scaling: 1 → No Change

Transformations on Lines (cont.)

- Rotating a line about an endpoint
  - Intent: P1 doesn't change, while P2 → P2'

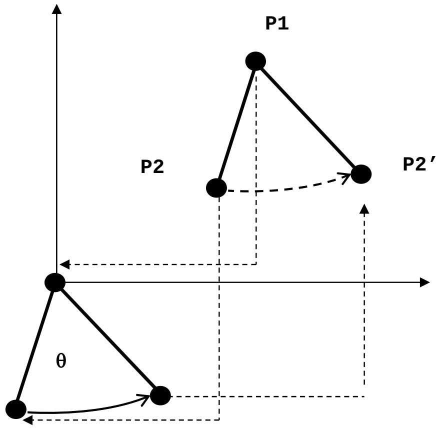
    (i.e. *rotate P2 by θ about P1*)
  - Again recall: rotation formulas are *about the origin*
    - What *is* the result of applying *Rotate (θ)* to P2 ?



Desired rotation
(rotate line P1-P2 about P1)

Transformations on Lines (cont.)

- Solution: as before – *force the rotation to be "about the origin"*



1. **P2.translate (-P1.x, -P1.y)**
2. **P2.rotate (θ)**
3. **P2.translate (P1.x, P1.y)**

Note "object-oriented" form

Transformations Using Matrices

- Translation

```
P = (x, y)

T = (+2, +3)

P′ = (x+2, y+3)
```

$$P' \;=\; \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 2 \\ 3 \end{bmatrix} \;=\; \begin{bmatrix} (x+2) \\ (y+3) \end{bmatrix}$$

Matrix Transformations (cont.)

- Rotation (CCW) about the origin

```
x′ =  x cos(θ)  -  y sin(θ)
y′ =  x sin(θ)  +  y cos (θ)
```

$$P' = \begin{bmatrix} x & y \end{bmatrix} * \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

$$= \begin{bmatrix} (x\cos(\theta) - y\sin(\theta)) & (x\sin(\theta) + y\cos(\theta)) \end{bmatrix}$$

Matrix Transformations (cont.)

- Scaling

$P = (x, y)$

$S = (s_x, s_y)$

$P' = (x*s_x, y*s_y)$

$$P' = \begin{bmatrix} x & y \end{bmatrix} * \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

$$= \begin{bmatrix} (x*s_x) & (y*s_y) \end{bmatrix}$$

Homogeneous Coordinates

- Motivation: uniformity between different matrix operations
- General Plan:
  - Represent a 2D point as a *triple* : **`[ x y 1 ]`**
  - Represent every transformation as a *3 x 3 matrix*
  - Use matrix ***multiplication*** for ***all*** transformations

Homogeneous Transformations

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix}$$

Translation

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Scaling

Applying Transformations

- Translation

$$\begin{bmatrix} x & y & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix} = \begin{bmatrix} (x+T_x) & (y+T_y) & 1 \end{bmatrix}$$

Applying Transformations (cont.)

- Rotation

$$[x \quad y \quad 1] * \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \left[\, (x\cos(\theta) - y\sin(\theta)) \quad (x\sin(\theta) + y\cos(\theta)) \quad 1 \,\right]$$

Applying Transformations (cont.)

- Scaling

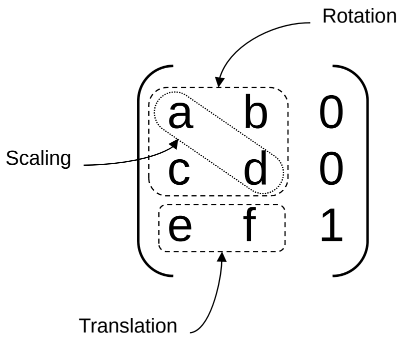
$$[x \quad y \quad 1] * \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} = \left[\, (x * S_x) \quad (y * S_y) \quad 1 \,\right]$$

Column-Major Representation

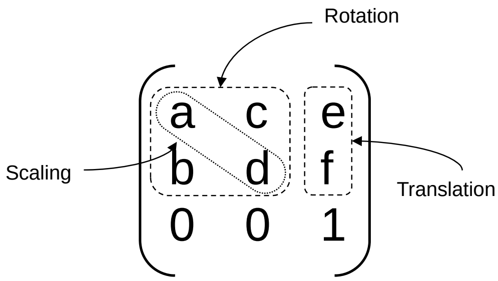
- Translation: $$\begin{bmatrix}(x+T_x)\\(y+T_y)\\1\end{bmatrix} = \begin{bmatrix}1 & 0 & T_x\\0 & 1 & T_y\\0 & 0 & 1\end{bmatrix} * \begin{bmatrix}x\\y\\1\end{bmatrix}$$
- Rotation: $$\begin{bmatrix}(x\cos(\theta)-y\sin(\theta))\\(x\sin(\theta)+y\cos(\theta))\\1\end{bmatrix} = \begin{bmatrix}\cos(\theta) & -\sin(\theta) & 0\\\sin(\theta) & \cos(\theta) & 0\\0 & 0 & 1\end{bmatrix} * \begin{bmatrix}x\\y\\1\end{bmatrix}$$
- Scaling: $$\begin{bmatrix}(x*S_x)\\(y*S_y)\\1\end{bmatrix} = \begin{bmatrix}S_x & 0 & 0\\0 & S_y & 0\\0 & 0 & 1\end{bmatrix} * \begin{bmatrix}x\\y\\1\end{bmatrix}$$

Active Matrix Areas

Row-major form



Column-major form



Same size “active area” – 6 elements (3x2 or 2x3)

Concatenation of Transforms

Typical Sequence:

P1 × Translate(tx,ty) = P2 ;

P2 × Rotate(θ) = P3 ;

P3 × Scale(sx,sy) = P4 ;

P4 × Translate(tx,ty) = P5 ;

Concatenation of Transforms (cont.)

- In (row-major) Matrix Form:

$$\begin{bmatrix} x1 & y1 & 1 \end{bmatrix} \times \begin{bmatrix} \text{Translate} \\ \text{(tx,ty)} \end{bmatrix} = \begin{bmatrix} x2 & y2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x2 & y2 & 1 \end{bmatrix} \times \begin{bmatrix} \text{Rotate}(\theta) \end{bmatrix} = \begin{bmatrix} x3 & y3 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x3 & y3 & 1 \end{bmatrix} \times \begin{bmatrix} \text{Scale} \\ \text{(sx,sy)} \end{bmatrix} = \begin{bmatrix} x4 & y4 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x4 & y4 & 1 \end{bmatrix} \times \begin{bmatrix} \text{Translate} \\ \text{(tx,ty)} \end{bmatrix} = \begin{bmatrix} x5 & y5 & 1 \end{bmatrix}$$

Concatenation of Transforms (cont.)

- Alternate Matrix Form:

$$\left(\left(\left(\left[\begin{matrix}x1 & y1 & 1\end{matrix}\right] \times \left[\begin{matrix}T1\end{matrix}\right]\right) \times \left[\begin{matrix}R1\end{matrix}\right]\right) \times \left[\begin{matrix}S1\end{matrix}\right]\right) \times \left[\begin{matrix}T2\end{matrix}\right]$$

$$= \left[\begin{matrix}x5 & y5 & 1\end{matrix}\right]$$

Concatenation of Transforms (cont.)

- Matrix multiplication is *associative* :

$$\left[\begin{matrix}x1 & y1 & 1\end{matrix}\right] \times \underbrace{\left(\left[\begin{matrix}T1\end{matrix}\right] \times \left[\begin{matrix}R1\end{matrix}\right] \times \left[\begin{matrix}S1\end{matrix}\right] \times \left[\begin{matrix}T2\end{matrix}\right]\right)} = \left[\begin{matrix}x5 & y5 & 1\end{matrix}\right]$$

$$\left[\begin{matrix}x1 & y1 & 1\end{matrix}\right] \times \left[\begin{matrix}M\end{matrix}\right] = \left[\begin{matrix}x5 & y5 & 1\end{matrix}\right]$$

In Column-Major Form

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} \textbf{\textit{Trans}} \\ \textbf{\textit{(x, y)}} \end{bmatrix} \times \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_3 \\ y_3 \\ 1 \end{bmatrix} = \begin{bmatrix} \textbf{\textit{Rot}}\ (\theta) \end{bmatrix} \times \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_4 \\ y_4 \\ 1 \end{bmatrix} = \begin{bmatrix} \textbf{\textit{Scale}} \\ \textbf{\textit{(sx, sy)}} \end{bmatrix} \times \begin{bmatrix} x_3 \\ y_3 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_5 \\ y_5 \\ 1 \end{bmatrix} = \begin{bmatrix} \textbf{\textit{Trans}} \\ \textbf{\textit{(x, y)}} \end{bmatrix} \times \begin{bmatrix} x_4 \\ y_4 \\ 1 \end{bmatrix}$$

Column-Major Form (cont.)

$$\begin{bmatrix} x_5 \\ y_5 \\ 1 \end{bmatrix} = \left(\begin{bmatrix} T2 \end{bmatrix} \times \left(\begin{bmatrix} S1 \end{bmatrix} \times \left(\begin{bmatrix} R1 \end{bmatrix} \times \left(\begin{bmatrix} T1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \right) \right) \right) \right)$$

$$\begin{bmatrix} x_5 \\ y_5 \\ 1 \end{bmatrix} = \underbrace{\left(\begin{bmatrix} T2 \end{bmatrix} \times \begin{bmatrix} S1 \end{bmatrix} \times \begin{bmatrix} R1 \end{bmatrix} \times \begin{bmatrix} T1 \end{bmatrix} \right)} \times \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_5 \\ y_5 \\ 1 \end{bmatrix} = \begin{bmatrix} M \end{bmatrix} \times \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

<< This page intentionally left (almost) blank >>

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming

John Clevenger / Scott Gordon

XIV – Applications of Affine Transforms

- **The Java `AffineTransform` Class**
- **Object (Local) Coordinate Systems**
- **Screen-Mapping Transforms**
- **Animation Transforms**
- **Hierarchical Object Transforms**
- **Dynamic Transforms**

CSc 133 Lecture Notes

XIV - Applications of Affine Transforms

Computer Science Dept.
CSUS

Overview

- The Java `AffineTransform` Class
- Object (Local) Coordinate Systems
- Screen-Mapping Transforms
- Animation Transforms
- Hierarchical Object Transforms
- The Graphics Transform Stack

`AffineTransform` Class

- `java.awt.geom.AffineTransform`
- Contains

  A 3×3 "transformation matrix"

  - Uses *column-major* form
  - *Only the active 2x3 elements can be accessed*

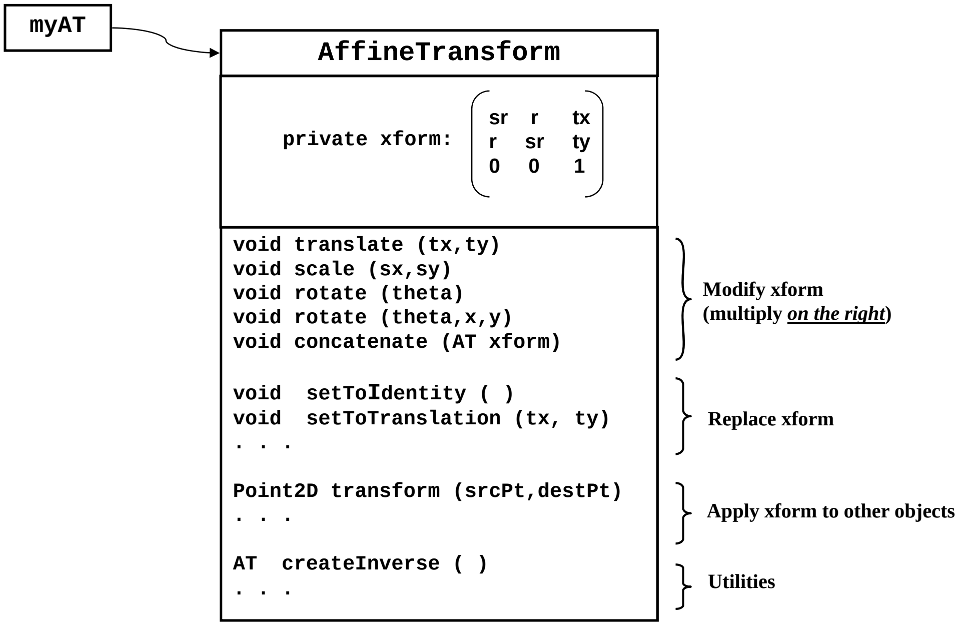
  Methods to *manipulate* the transformation matrix

  Methods to *apply* the transform to other objects

- Construction example:

```
AffineTransform myAT = new AffineTransform();
```

`AffineTransform` ("AT") Objects



$$\begin{pmatrix} sr & r & tx \\ r & sr & ty \\ 0 & 0 & 1 \end{pmatrix}$$

```
void translate (tx,ty)
void scale (sx,sy)
void rotate (theta)
void rotate (theta,x,y)
void concatenate (AT xform)

void  setToIdentity ( )
void  setToTranslation (tx, ty)
. . .

Point2D transform (srcPt,destPt)
. . .

AT  createInverse ( )
. . .
```

Using An "AT" Object

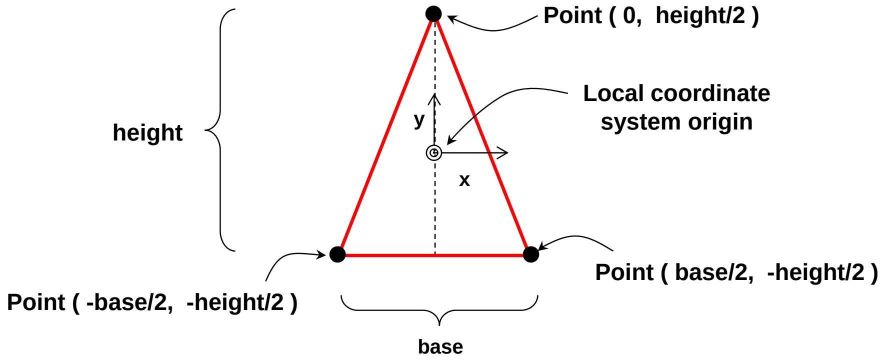
```
...
Point p1 = new Point(x,y);
Point p2 = new Point();
AffineTransform myAT = new AffineTransform();
myAT.rotate(Math.toRadians(45));
myAT.transform (p1,p2);
```

$$\begin{bmatrix} x2 \\ y2 \\ 1 \end{bmatrix} = \begin{bmatrix} \quad & & \\ & Rotate(45^0) & \\ & & \quad \end{bmatrix} \times \begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix}$$

"Local" Coordinate Systems

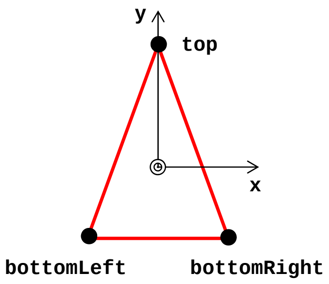
Define objects *relative to their own origin*

- Example: triangle
  - Base & Height
  - Local origin at "center"
  - Vertices defined *relative to local origin*


Point (0, height/2)
Local coordinate system origin
height
y
x
Point (base/2, -height/2)
Point (-base/2, -height/2)
base

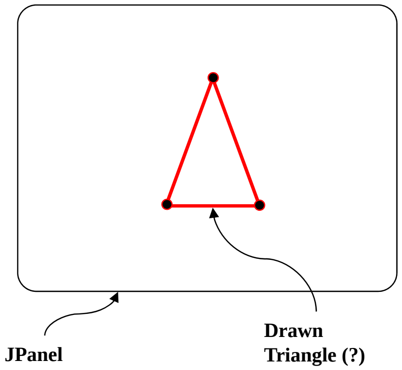
Triangle Class

```
/** This class defines an isosceles triangle with a specified base and height.
 *  The triangle coordinates are defined in “local space”, and the local
 *  space axis orientation is X to the right and Y upward.
 */
public class Triangle {
  private Point top, bottomLeft, bottomRight ;
  private Color color ;
  public Triangle (int base, int height) {
    top = new Point (0, height/2);
    bottomLeft = new Point (-base/2, -height/2);
    bottomRight = new Point (base/2, -height/2);
    color = Color.red ;
  }
  public void draw (Graphics g) {
    g.setColor(color);
    g.drawLine (top.x, top.y, bottomLeft.x, bottomLeft.y) ;
    g.drawLine (bottomLeft.x, bottomLeft.y, bottomRight.x, bottomRight.y) ;
    g.drawLine (bottomRight.x, bottomRight.y, top.x, top.y) ;
  }
}
```



Drawing A Triangle

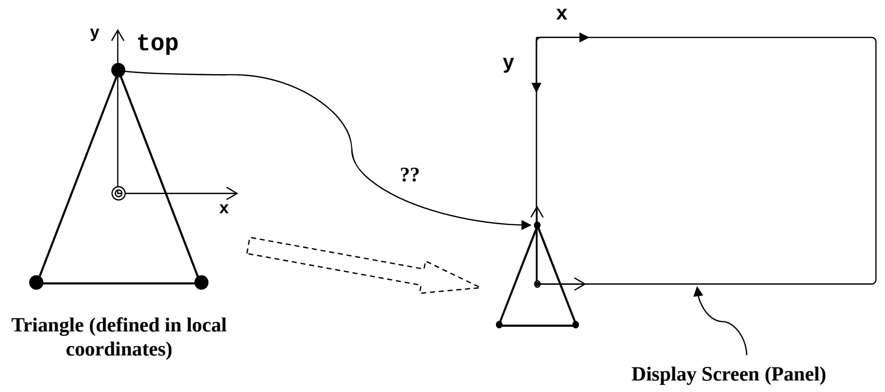
```
/** This class defines a panel containing a triangle.
 *  Repainting the panel causes the triangle to be drawn.
 */
public class DisplayPanel extends JPanel {
  private Triangle myTriangle ;
  public DisplayPanel () {
    myTriangle = new Triangle (100, 100) ;
  }
  public void paintComponent (Graphics g) {
    super.paintComponent(g);
    myTriangle.draw(g) ;
  }
}
```



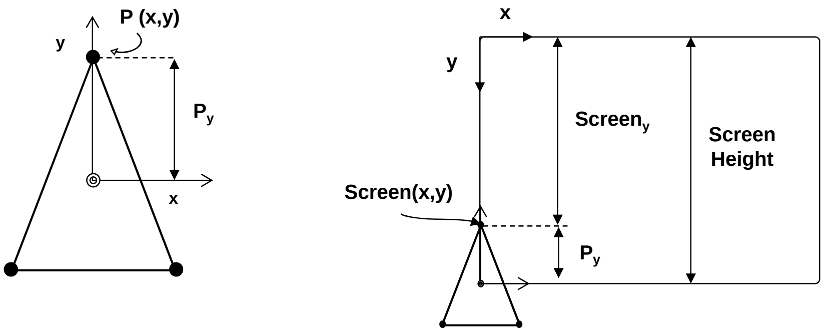
Run

Mapping To Screen Location

- Suppose desired location was “centered at lower-left screen corner”
- How do we compute location of “top”?



Mapping To Screen Location (cont.)



- $Screen_x = P_x$
- $Screen_y = ScreenHeight - P_y$

$$= \underbrace{(-1*(P_y))}_{\texttt{Scale}_y\texttt{(-1)}} + \underbrace{ScreenHeight}_{\texttt{Translate}_y\texttt{(ScreenHeight)}}$$

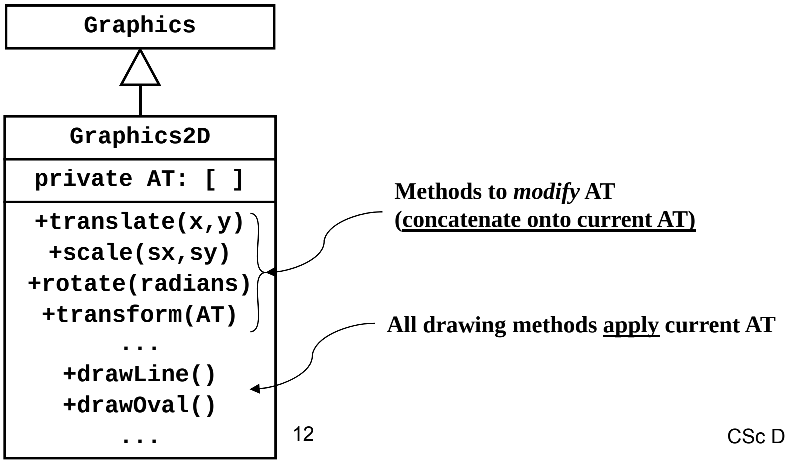
Applying the Screen Transform

```
/** This class draws an Isosceles Triangle applying a “screen mapping”
 *  transformation to the triangle points.
 */
public class Triangle {

  private Point top, bottomLeft, bottomRight ;
  private Color color ;
  ...

  public void draw (Graphics g) {
    // create an AT to transform triangle points to “screen space”
    AffineTransform screenXform = new AffineTransform();
    screenXform.translate (0, panel.getHeight());
    screenXform.scale (1, -1);

    // apply the screen transform to the triangle points
    screenXform.transform(top,top);
    screenXform.transform(bottomLeft,bottomLeft);
    screenXform.transform(bottomRight,bottomRight);

    // draw the (transformed) triangle
    g.setColor(color);
    g.drawLine (top.x, top.y, bottomLeft.x, bottomLeft.y) ;
    g.drawLine (bottomLeft.x, bottomLeft.y, bottomRight.x, bottomRight.y) ;
    g.drawLine (bottomRight.x, bottomRight.y, top.x, top.y) ;
  }
}
```

Run

The **Graphics2D** Class

- A subclass of **`Graphics`**
- **`paintComponent()`** actually receives a **`Graphics2D`**
- Every **`Graphics2D`** contains an **`AT`**
  - The **`AT`** is applied to all coordinates during drawing



Methods to *modify* AT (concatenate onto current AT)

All drawing methods apply current AT

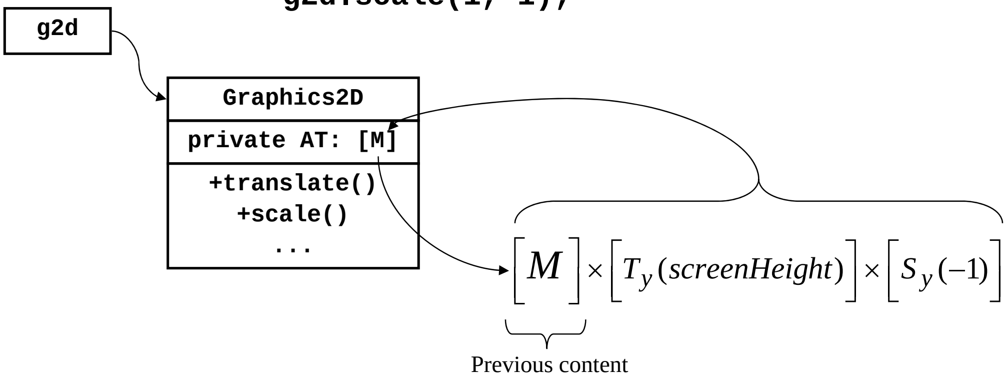
Using Graphics2D's AT

```
/** Repainting the panel applies a scale and translate to the AffineTransform
 *  in the Graphics2D object, then tells the triangle to draw itself using
 *  that G2D object.  This causes the specified scale and translate to be
 *  applied when the triangle it drawn.
 */
public class DisplayPanel extends JPanel {
  private Triangle myTriangle ;
  public DisplayPanel () {
    myTriangle = new Triangle (100, 100) ;
  }
  public void paintComponent (Graphics g) {
    super.paintComponent(g);
    //convert the incoming Graphics to a Graphics2D
    Graphics2D g2d = (Graphics2D) g ;
    //apply a translation of ScreenHeight
    g2d.translate (0, this.getHeight());
    //apply a scale of -1 in Y
    g2d.scale(1,-1);
    //draw the triangle using the Graphics2D object
    //  (thus applying the screen transformation)
    myTriangle.draw(g2d) ;
  }
}
```

Using Graphics2D's AT (cont.)

- Effect of modifying **g2d**'s transform in **paintComponent()** :

```
g2d.translate (0, this.getHeight());
g2d.scale(1,-1);
```



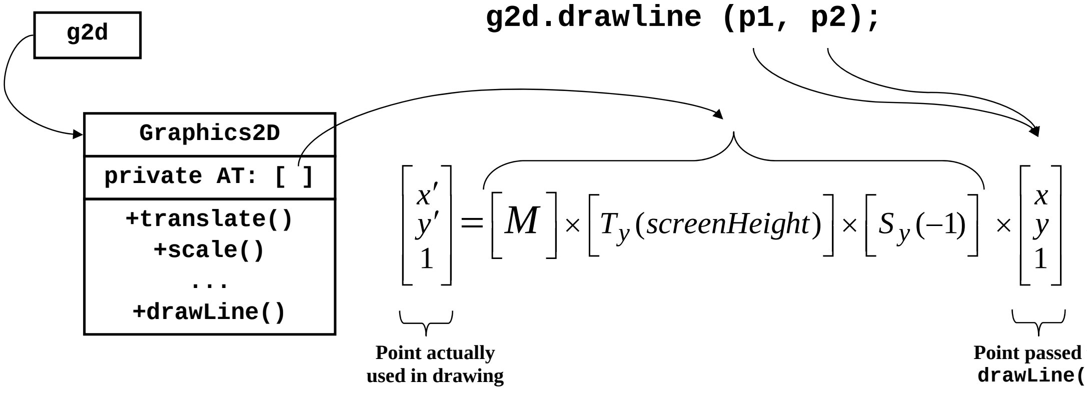
$$\underbrace{\begin{bmatrix} M \end{bmatrix}}_{\text{Previous content}} \times \overbrace{\begin{bmatrix} T_y(screenHeight) \end{bmatrix} \times \begin{bmatrix} S_y(-1) \end{bmatrix}}$$

Using Graphics2D's AT (cont.)

```
/** This class defines a triangle, as before.  The only difference is that
 *  now the triangle's draw() method is receiving a Graphics2D object.
 *  The Graphics2D object applies its current AffineTransform to all
 *  coordinates prior to performing any output operation.
 */
public class Triangle {
  private Point top, bottomLeft, bottomRight ;
  private Color color ;
  public Triangle (int base, int height) {
    top = new Point (0, height/2);
    bottomLeft = new Point (-base/2, -height/2);
    bottomRight = new Point (base/2, -height/2);
    color = Color.red ;
  }
  public void draw (Graphics2D g2d) {
    g2d.setColor(color);
    g2d.drawLine (top.x, top.y, bottomLeft.x, bottomLeft.y) ;
    g2d.drawLine (bottomLeft.x, bottomLeft.y, bottomRight.x, bottomRight.y) ;
    g2d.drawLine (bottomRight.x, bottomRight.y, top.x, top.y) ;
  }
}
```

Using Graphics2D's AT (cont.)

- Effect of using **g2d** to draw a line in **Triangle.draw()**:



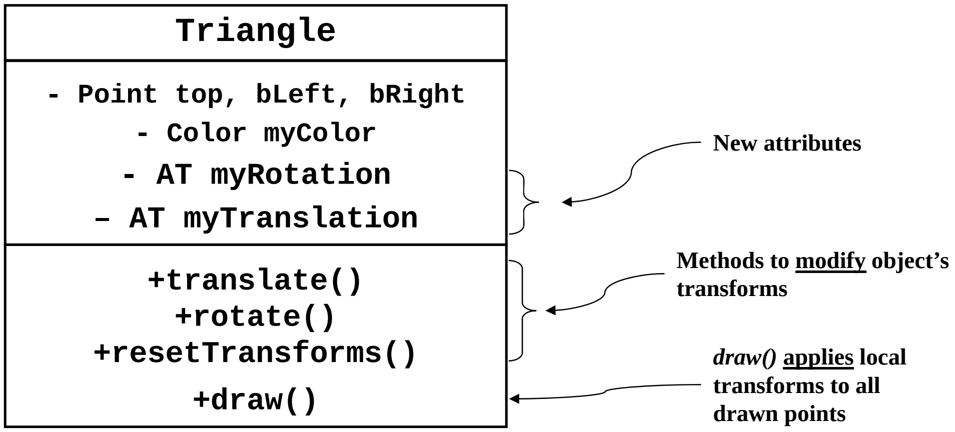
g2d.drawline (p1, p2);

$$\underbrace{\begin{bmatrix}x'\\y'\\1\end{bmatrix}}_{\text{Point actually used in drawing}} = \overbrace{\begin{bmatrix}M\end{bmatrix}\times\begin{bmatrix}T_y(screenHeight)\end{bmatrix}\times\begin{bmatrix}S_y(-1)\end{bmatrix}}\times\underbrace{\begin{bmatrix}x\\y\\1\end{bmatrix}}_{\text{Point passed to drawLine()}}$$

Run

Transformable Objects

- Expand objects to contain *"local transforms"*
- Arrange to *apply an object's transforms* when it is drawn



```
/** This class defines a triangle with an internal AffineTransform (AT). Client code
 *  can apply arbitrary transformations to the triangle by invoking methods to
 *  update/modify the internal ATs; when the triangle is drawn it automatically
 *  applies its current AT to output coordinates. */
public class Triangle {
  private Point top, bottomLeft, bottomRight ;
  private Color myColor ;
  private AffineTransform myRotation, myTranslation ;
  public Triangle (int base, int height) {
    top = new Point (0, height/2);
    bottomLeft = new Point (-base/2, -height/2);
    bottomRight = new Point (base/2, -height/2);
    myColor = Color.red ;
    myRotation = new AffineTransform();
    myTranslation = new AffineTransform();
  }
  public void rotate (double radians) {
    myRotation.rotate (radians);
  }
  public void translate (double dx, double dy) {
    myTranslation.translate (dx, dy);
  }
  public void resetTransform() {
    myRotation.setToIdentity();
    myTranslation.setToIdentity();
  }
  //...continued...
```

Transformable Objects (cont.)

```
// ... Triangle class, cont.

/** This method applies the triangle’s AT to the received Graphics2D object, then uses
  * that object (with the additional transformations) to draw the triangle.
 */
public void draw (Graphics2D g2d) {
  // set the drawing color for the triangle
  g2d.setColor(myColor);

  // append the triangle’s transforms to the AT in the Graphics2D object.  Note that
  // translation is appended first, on the right, so rotation gets performed first –
  // hence rotation is about the “local origin”
  g2d.transform(myTranslation);
  g2d.transform(myRotation);

  // draw the triangle, using the g2d to apply the triangle’s transformations
  g2d.drawLine (top.x, top.y,  bottomLeft.x, bottomLeft.y);
  g2d.drawLine (bottomLeft.x, bottomLeft.y,  bottomRight.x, bottomRight.y);
  g2d.drawLine (bottomRight.x, bottomRight.y,  top.x, top.y);
}

} //end of Triangle class
```

```
/** This class defines a panel containing a triangle.  It applies a
 *  simple set of transformations to the triangle (by calling the triangle’s
 *  transformation methods when the triangle is created).  The panel’s
 *  paintComponent() method applies the “screen coordinate” transformation
 *  to the Graphics2D object, and tells the triangle to “draw itself”.
 *  The triangle applies its transformations to the Graphics2D object
 *  in its draw() method.
 */
public class DisplayPanel extends JPanel {
  private Triangle myTriangle ;
  public DisplayPanel () {
    myTriangle = new Triangle (50, 100) ;      //construct a Triangle
    myTriangle.translate (200, 200);           //apply some transformations
    myTriangle.rotate (Math.toRadians(90));    // to the triangle
  }
  public void paintComponent (Graphics g) {
    super.paintComponent(g);

    //apply the “Screen coordinate” transformation to the Graphics2D object
    Graphics2D g2d = (Graphics2D) g ;
    g2d.translate (0, this.getHeight());
    g2d.scale(1,-1);

    //tell the triangle to draw itself
    myTriangle.draw(g2d) ;
  }
}
```

Composite g2d Transforms

- Transformations applied to triangle vertices:

$$\underset{\text{Drawn vertex}}{\underset{\uparrow}{\begin{bmatrix}x'\\y'\\1\end{bmatrix}}} = \underbrace{\begin{bmatrix}M\end{bmatrix}}_{\text{Original g2D AT contents}} \times \underbrace{\begin{bmatrix}T_{screen}\end{bmatrix}\times\begin{bmatrix}S_{screen}\end{bmatrix}}_{\text{Screen transformation}} \times \underbrace{\begin{bmatrix}T_{tri}\end{bmatrix}\times\begin{bmatrix}R_{tri}\end{bmatrix}}_{\text{Triangle transformation}} \times \underset{\text{Triangle vertex}}{\underset{\uparrow}{\begin{bmatrix}x\\y\\1\end{bmatrix}}}$$

$\longleftarrow$

Order of application of transformations

Run

More on *Transformation Order*

- Suppose an interactive program implements:

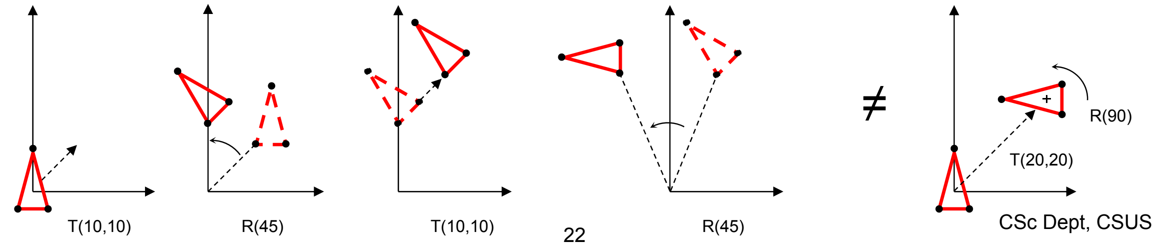
  Click = translate (10,10),   Drag = rotate (± 5°)

- Consider the interactive sequence

  Drag (45°), Click, Drag (45°), Click

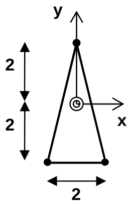
- Expected result:
  - Rotation by a total of 90°, Translation by a total of (20,20)

- Actual sequence: $[R(45)]\times[T(10{,}10)]\times[R(45)]\times[T(10{,}10)]\times\begin{bmatrix}X\\Y\\1\end{bmatrix}$

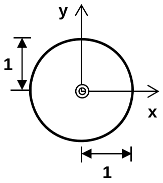


Hierarchical Objects

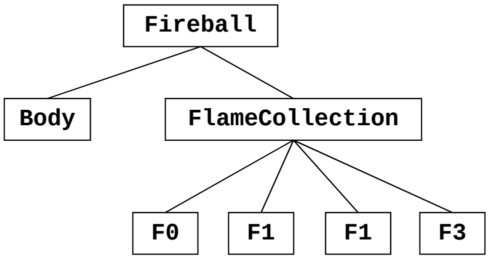
- We can build an object by combining
  - Simpler “parts”
  - Transformations to “orient” the parts



A “Flame” object



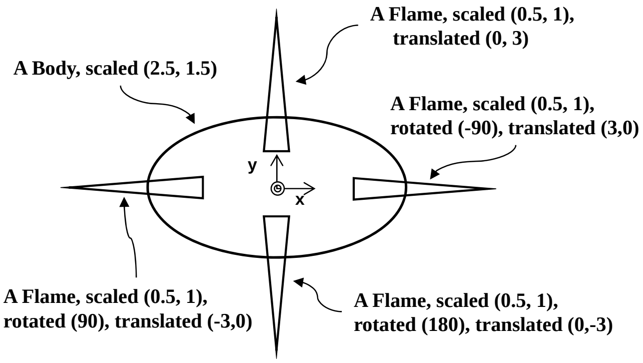
A “Body” object



A hierarchical “Fireball” object

Hierarchical Objects (cont.)

- Fireball Transformations


A Flame, scaled (0.5, 1),
translated (0, 3)
A Body, scaled (2.5, 1.5)
A Flame, scaled (0.5, 1),
rotated (-90), translated (3,0)
y
x
A Flame, scaled (0.5, 1),
rotated (90), translated (-3,0)
A Flame, scaled (0.5, 1),
rotated (180), translated (0,-3)

A hierarchical “Fireball” object

Hierarchical Objects (cont.)

```
/** Defines a single "flame" to be used as an arm of a fireball.
 *  The Flame is modeled after the "Triangle" class, but specifies
 *  fixed dimensions of 2 (base) by 4 (height) in local space.
 *  Clients using the Flame can scale it to have any desired proportions.
 */
public class Flame {
  private Point top, bottomLeft, bottomRight ;
  private Color myColor ;
  private AffineTransform myTranslation ;
  private AffineTransform myRotation ;
  private AffineTransform myScale ;

  public Flame (){
    // define a default flame with base=2, height=4, and origin in the center.
    top = new Point (0, 2);
    bottomLeft = new Point (-1, -2);
    bottomRight = new Point (1, -2);
    // initialize the transformations applied to the Flame
    myTranslation = new AffineTransform();
    myRotation = new AffineTransform();
    myScale = new AffineTransform();
  }

  //...continued
```

```
  // Flame class, continued...
  public void rotate (double degrees)     {
    myRotation.rotate (Math.toRadians(degrees));
  }
  public void scale (double sx, double sy) {
    myScale.scale (sx, sy);
  }
  public void translate (double dx, double dy) {
    myTranslation.translate (dx, dy);
  }
  public void draw (Graphics2D g2d) {
    // save the current graphics transform for later restoration
    AffineTransform saveAT = g2d.getTransform() ;
    // Append this shape's transforms to the graphics object's transform. Note the
    // ORDER: Translation will be done FIRST, then Scaling, and lastly Rotation
    g2d.transform(myRotation);
    g2d.transform(myScale);
    g2d.transform(myTranslation);          ← Translation is applied FIRST
    // draw this object in the defined color
    g2d.setColor(myColor);
    g2d.drawLine (top.x, top.y, bottomLeft.x, bottomLeft.y) ;
    g2d.drawLine (bottomLeft.x, bottomLeft.y, bottomRight.x, bottomRight.y);
    g2d.drawLine (bottomRight.x, bottomRight.y, top.x, top.y) ;
    // restore the old graphics transform (remove this shape's transform)
    g2d.setTransform (saveAT) ;
  }
} // end of Flame class
```

```
/** Defines a "Body" for a fireball; the "body" is just a scalable
 *  circle of radius=1 with its origin in the center.
 */
public class Body {
  private int myRadius;
  private Color myColor ;
  private AffineTransform myTranslation ;
  private AffineTransform myRotation ;
  private AffineTransform myScale ;
  public Body () {
    myRadius = 1;  myColor = Color.yellow ;
    myTranslation = new AffineTransform();
    myRotation = new AffineTransform();
    myScale = new AffineTransform();
  }
  // ...code here implementing rotate(), scale(), and translate() as in the Flame class
  public void draw (Graphics2D g2d) {
    AffineTransform saveAT = g2d.getTransform() ;
    g2d.transform(myTranslation);          ← This time translation is applied LAST
    g2d.transform(myRotation);
    g2d.transform(myScale);
    g2d.setColor(myColor);
    // calculate the local-space location of the lower-left corner of the bounding box
    // for the circle, since that's where the object will be drawn from in local space
    Point boxCorner = new Point (-myRadius, -myRadius);
    g2d.fillOval (boxCorner.x, boxCorner.y, myRadius*2, myRadius*2) ;
    g2d.setTransform (saveAT) ;
  }
}
```

```
/** This class defines a "Fireball", which is a hierarchical object composed
 *  of a scaled "Body" and four scaled, rotated, and translated "Flames".
 */
public class Fireball {
  private Body myBody ;
  private Flame [] flames ;
  private AffineTransform myTranslation, myRotation, myScale ;
  public Fireball () {
    myTranslation = new AffineTransform() ;
    myRotation = new AffineTransform() ;
    myScale = new AffineTransform() ;
    myBody = new Body();           // create a properly-scaled Body for the Fireball
    myBody.scale(2.5, 1.5);
    flames = new Flame [4];        // create an array to hold the four flames
    // create four flames, each translated “up” in Y and then scaled, and rotated into
    // position relative to the Body's origin
    Flame f0 = new Flame();  f0.translate(0, 4);   f0.scale (0.5, 0.8);
    flames[0] = f0 ;   f0.setColor(Color.red);
    Flame f1 = new Flame(); f1.translate(0, 7);f1.rotate(-90);f1.scale(0.5, 0.5);
    flames[1] = f1 ;   f1.setColor(Color.green);
    Flame f2 = new Flame(); f2.translate(0, 4);f2.rotate(180);f2.scale(0.5, 0.8);
    flames[2] = f2 ;   f2.setColor(Color.blue);
    Flame f3 = new Flame(); f3.translate(0, 7);f3.rotate(90);f3.scale(0.5, 0.5);
    flames[3] = f3;    f3.setColor(Color.magenta);
  }
  // continued...
```

Hierarchical Objects (cont.)

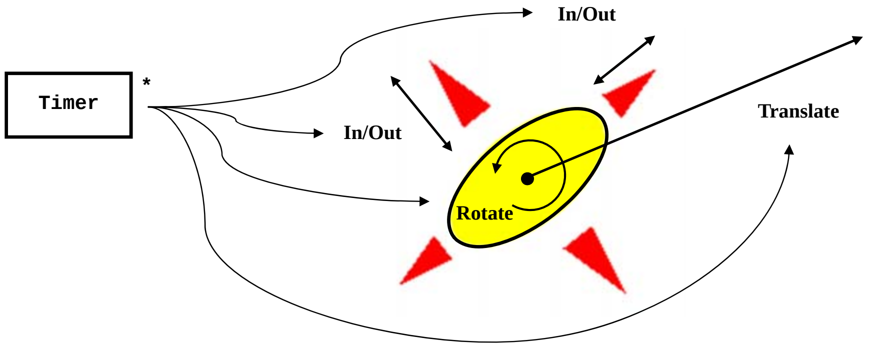
```
  // Fireball class, continued...
  // ...code here implementing rotate(), scale(), and translate() as in the Flame class
  // This method applies the Fireball's current attributes to the g2d object
  //   and then draws each of the Fireball components

  public void draw (Graphics2D g2d) {
    // save the current graphics transform for later restoration
    AffineTransform saveAT = g2d.getTransform() ;
    // append this shape's transforms to the graphics object's transform
    g2d.transform(myTranslation);
    g2d.transform(myRotation);
    g2d.transform(myScale);
    // draw this shape's components, modified by the updated g2d transformation
    myBody.draw(g2d);
    for (Flame f : flames) {
      f.draw(g2d);
    }
    // restore the old graphics transform (remove this shape's transform)
    g2d.setTransform (saveAT) ;
  }
} //end of Fireball class
```

```
/** This class displays a "Fireball" object, translating and rotating it into
 *  position on the screen, and telling it to draw itself (including drawing
 *  any sub-components which it contains).
 */
public class DisplayPanel extends JPanel {
  Fireball myFireball ;
  public DisplayPanel () {
    // create a Fireball to display
    myFireball = new Fireball ();
    // rotate, scale, and translate this Fireball on the panel
    myFireball.translate (300, 200) ;
    myFireball.rotate (45) ;
    myFireball.scale(20,20);
  }
  public void paintComponent (Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D) g ;

    // map object coordinates to screen
    g2d.translate (0, this.getHeight());
    g2d.scale (1, -1);

    // tell the fireball to draw itself (and its sub-components)
    myFireball.draw(g2d) ;
  }
}
```

Run

Dynamic Transformations

- We can alter an object’s transforms “on-the-fly”
  - Vary sub-component transforms relative to parent
  - Vary entire object transform



Dynamic Transformations (cont.)

```
/** This class defines a Panel containing a “Fireball” object and uses a
 *  Timer to modify the Fireball transformations (position and orientation).
 */
public class DisplayPanel extends JPanel implements ActionListener {
   Fireball myFireball ;
   public DisplayPanel () {
      // create a Fireball to display
      myFireball = new Fireball ();
      myFireball.scale(20,20);
      // start a Timer to periodically update the fireball
      Timer theTimer = new Timer (10, this) ;
      theTimer.start() ;
   }
   public void actionPerformed (ActionEvent e) {
      myFireball.update() ;
      this.repaint() ;
   }
   public void paintComponent (Graphics g) {
      super.paintComponent(g);
      Graphics2D g2d = (Graphics2D) g ;
      // map object coordinates to screen
      g2d.translate (0, this.getHeight());
      g2d.scale (1, -1);
      // tell the fireball to draw itself (and its sub-components)
      myFireball.draw(g2d) ;
   }
}
```

Dynamic Transformations (cont.)

```
/** This class defines a Fireball object which supports dynamic alteration
 *  of both the Fireball position & orientation, and also of the offset of
 *  the flames from the Fireball body.
 */
public class Fireball {
    //...declarations here for Body, Flames, and fireball transforms, as before;
    // and code here to define the fireball body and flames, and to define
    // methods for applying transformations, as before...
    private double flameOffset = 0 ;         // current flame distance from fireball
    private double flameIncrement = +0.05 ; // change in flame distance each tick
    private double maxFlameOffset = 0.5 ;   // max distance before reversing

    // Invoked to update the fireball (and its sub-component's) transformations
    public void update () {
        // update the fireball position and orientation
        myTranslation.translate(1,1);
        myRotation.rotate(Math.toRadians(1)) ;
        // update the flame positions (move them along their local Y axis)
        flameOffset += flameIncrement ;
        for (Flame f:flames) {
            f.translate (0, flameOffset);  // this is why flames are TRANSLATED 1st
        }
        // reverse direction of flame movement for next time if we've hit the max
        if (Math.abs(flameOffset) >= maxFlameOffset) {
            flameIncrement *= -1 ;
        }
    }
```

Dynamic Transformations (cont.)

```
    // Fireball class, continued...
    /** Draw the Fireball and its sub-components, applying the current
     *  shape transformation
     */
    public void draw (Graphics2D g2d) {

        // save the current graphics transform for later restoration
        AffineTransform saveAT = g2d.getTransform() ;

        // append this shape's transforms to the graphics object's transform
        g2d.transform(myTranslation);
        g2d.transform(myRotation);
        g2d.transform(myScale);

        // draw this shape's components, modified by the updated g2d transformation
        myBody.draw(g2d);

        for (Flame f : flames) {    // draw each Flame.  Recall that each Flame
            f.draw(g2d);            // applies its translation first, causing it to
        }                           // move "up and down" on its own Y axis

        // restore the old graphics transform (remove this shape's transform)
        g2d.setTransform (saveAT) ;
    }
} // end Fireball class
```

Dynamic Transformations (cont.)

```
/** Defines a single “Flame" to be used as an arm of a fireball;
 *  uses “drawPolygon()” to draw a FILLED triangle.
 */
public class Flame {
    // ...code here to construct the Flame and provide methods for applying
    //   transformations, as before...
    public void draw (Graphics2D g2d) {
        // save the current graphics transform for later restoration
        AffineTransform saveAT = g2d.getTransform() ;
        // append this shape's transforms to the graphics object's transform
        g2d.transform(myRotation);
        g2d.transform(myScale);
        g2d.transform(myTranslation);        // note, Flames “move” before other xforms
        // draw this object in the defined color
        g2d.setColor(myColor);
        int [] xPts = new int [] {top.x, bottomLeft.x, bottomRight.x} ;
        int [] yPts = new int [] {top.y, bottomLeft.y, bottomRight.y} ;
        g2d.fillPolygon (xPts, yPts, 3);   // Flame’s vertices define a filled polygon
        // restore the old graphics transform (remove this shape's transform)
        g2d.setTransform (saveAT) ;
    }
}
```

Run

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

XV – Viewing Transformations

- **The World Coordinate System**
- **World Windows**
- **The Normalized Device (ND)**
- **World to ND transform**
- **Viewing Transformation Matrix (VTM)**
- **2D Viewing Operations (Zoom and Pan)**
- **Mapping the Mouse to World Coordinates**
- **Clipping**
- **Cohen-Sutherland Algorithm**

CSc 133 Lecture Notes

XV - Viewing Transformations

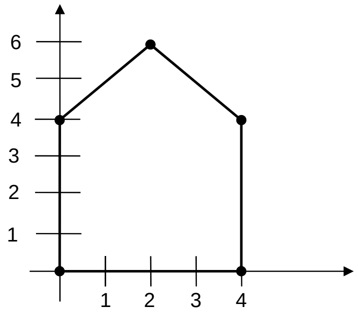
Computer Science Dept.
CSUS

Overview

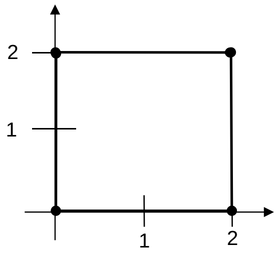
- The World Coordinate System
- Mapping From World to Screen Coordinates
  - World Windows, Normalized Devices, Screen Viewports, and the Viewing Transformation Matrix
- 2D Viewing Operations (Zoom and Pan)
- Mapping User Input to World Coordinates
- Clipping and the Cohen-Sutherland Algorithm

Local Coordinate Systems

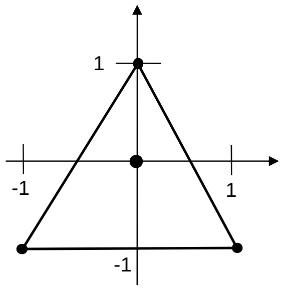
- Each object is defined in its “own space”



Pentagon

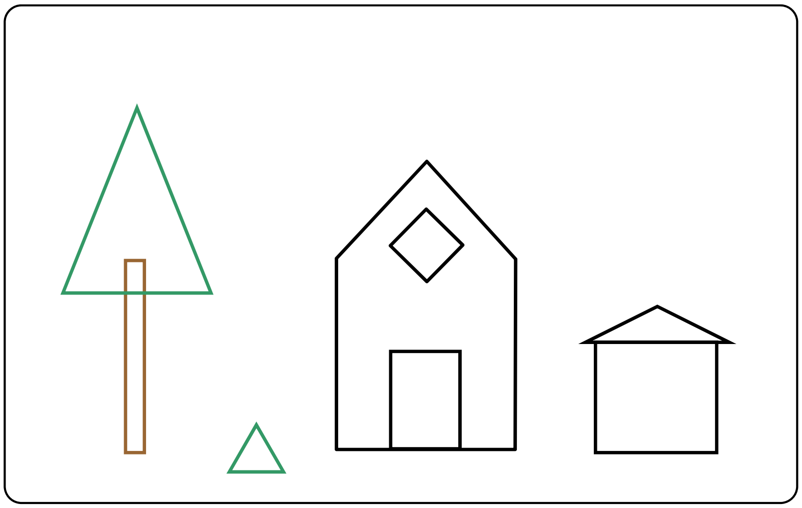


Box



Triangle

Creating A “World”



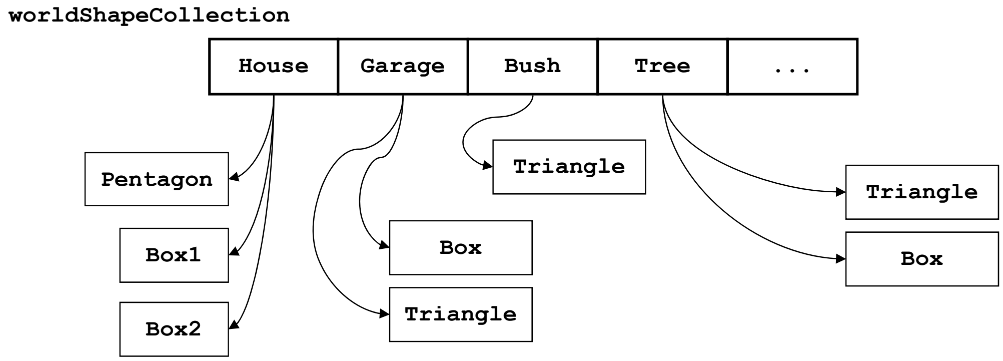
Tree
(Triangle + Box)

Bush
(Triangle)

House
(Pentagon + Two Boxes)

Garage
(Box + Triangle)

The World Object Collection



```
DrawingPanel.paintComponent(Graphics g){
  save current graphics state;
  apply "screen transform" to g;
  for (Shape s : worldShapeCollection){
    s.draw(g);
  }
  restore graphics state;
}
```

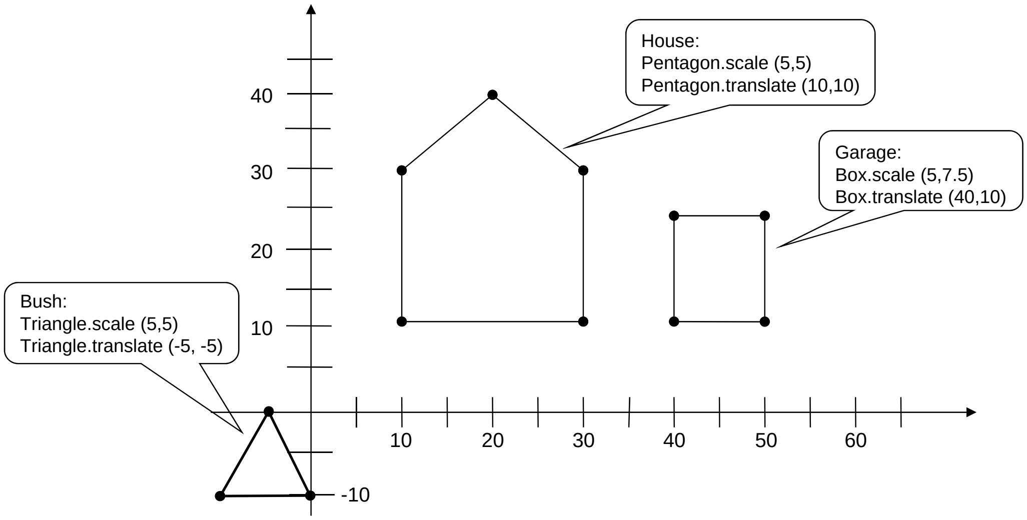
```
Shape.draw(Graphics g)  {
  save current graphics state;
  apply myTransforms to g ;
  draw myself using g;
  for (each child shape)  {
    child.draw(g);
  }
  restore graphics state;
}
```

The World Coordinate System

- "World" ("virtual" or "user") units
  - Independent of display device
  - Can represent inches, feet, meters…
- Infinite in all directions
- Object instances are "placed" in the World via *transformations*

World Coordinate System (cont.)

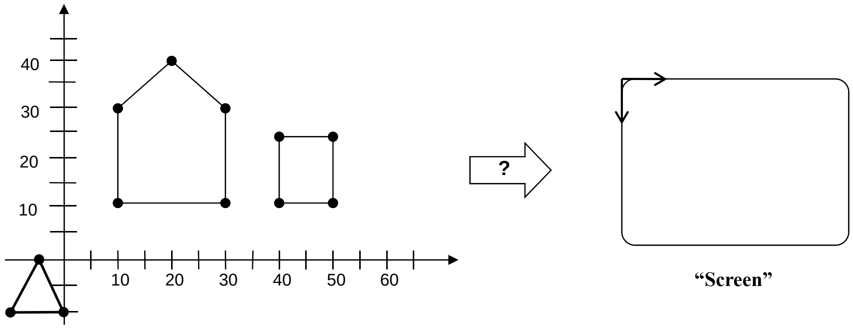
Example:



Drawing The World On The Screen

Needed:

- A way to determine what portion of the (infinite) World gets drawn on the (finite) screen
- A "mapping" or *transformation* from *World* to *Screen* coords



Drawing The World (cont.)

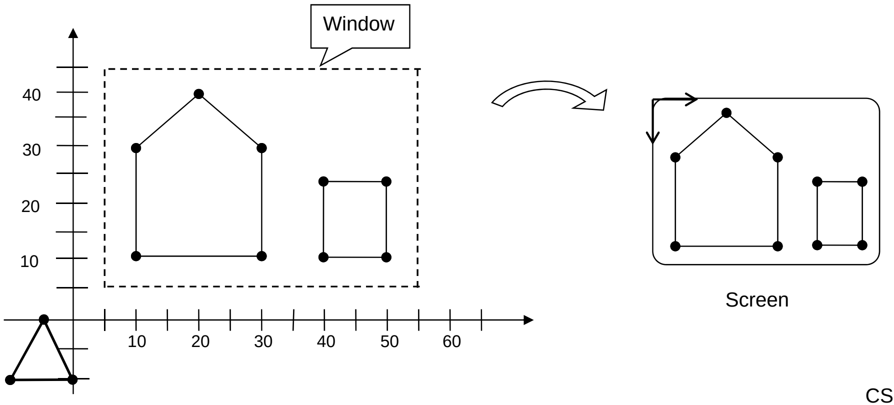
Solution:

- The “Virtual (World) Window”
- A *two-step* mapping through a “*Normalized Device*”

The World “Window”

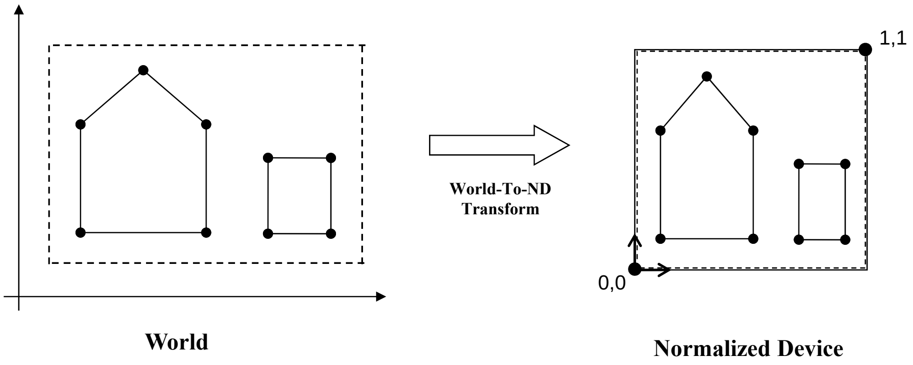
Defines the part of the world that appears on screen

- Corners of the window match the corners of the screen (“viewport”)
- Objects inside window are positioned proportionally in the viewport
- Objects outside window are “clipped” (discarded)



The “Normalized Device”

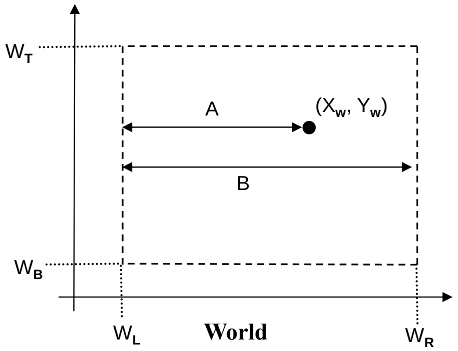
- Properties of the Normalized Device (ND):
  - Square
  - Fractional Coordinates (0.0 .. 1.0)
  - Origin at Lower Left
  - Corners correspond to world window



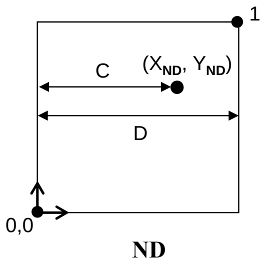
World-To-ND Transform

Consider a single point’s X coordinate

- Need to achieve proportional positioning on the ND



$$\frac{A}{B} = \frac{C}{D}$$



$$A = X_w - W_L \;; \qquad B = W_R - W_L \;; \qquad D = 1;$$

$$\therefore \; C = X_{ND} = \frac{(X_w - W_L)}{(W_R - W_L)} = (X_w - W_L) * \frac{1}{(W_R - W_L)}$$

World-To-ND Transform (cont.)

Consider the form of X_{ND} :

$$X_{ND} = \underbrace{(X_w - W_L)}_{\substack{\textbf{A } \underline{\textit{translation}} \\ \textit{(by WindowLeft)}}} * \underbrace{\frac{1}{(W_R - W_L)}}_{\substack{\textbf{A } \underline{\textit{scale}} \textit{ (by} \\ \textit{1/windowWidth)}}}$$

Similar rules can be used to derive Y_{ND} :

$$Y_{ND} = \underbrace{(Y_w - W_B)}_{\underline{\textit{Translation}}} * \underbrace{\frac{1}{(W_T - W_B)}}_{\underline{\textit{Scale}}}$$

World-To-ND Transform (cont.)

X_{ND} = (X_W • Translate (-W_L)) • Scale (1 / WindowWidth)

Y_{ND} = (Y_W • Translate (-W_B)) • Scale (1 / WindowHeight)

or

P_{ND} = (P_W • Translate (-W_L, -W_B)) • Scale (1/WindowWidth, 1/WindowHeight)

- In Matrix Form:

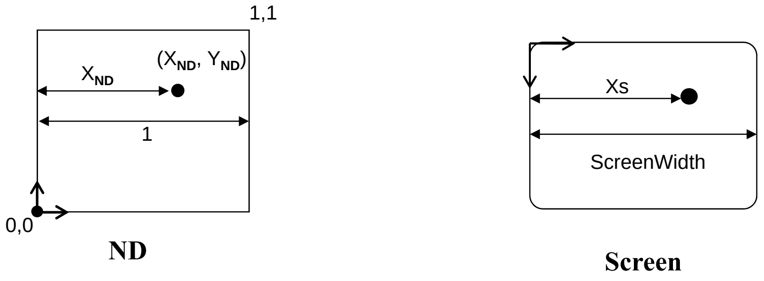
$$\begin{bmatrix} X_{ND} \\ Y_{ND} \\ 1 \end{bmatrix} = \left(\begin{bmatrix} 1/W_w & 0 & 0 \\ 0 & 1/W_h & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -W_L \\ 0 & 1 & -W_B \\ 0 & 0 & 1 \end{bmatrix} \right) \begin{bmatrix} X_W \\ Y_W \\ 1 \end{bmatrix}$$

X

$$\begin{bmatrix} X_{ND} \\ Y_{ND} \\ 1 \end{bmatrix} = \begin{pmatrix} \textbf{World-to-} \\ \textbf{Normalized-} \\ \textbf{Device} \\ \textbf{(W2ND)} \\ \textbf{Transform} \end{pmatrix} \begin{bmatrix} X_W \\ Y_W \\ 1 \end{bmatrix}$$

ND-To-Screen Transform

- A similar approach can be applied

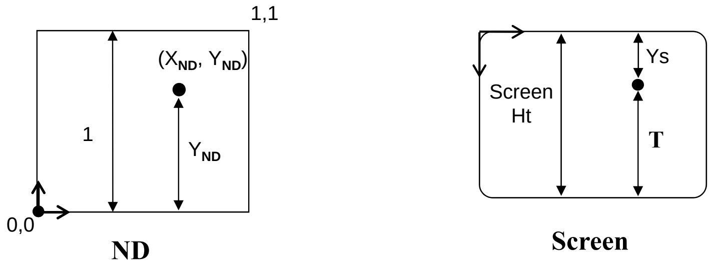


$$\frac{X_{ND}}{1} = \frac{X_S}{ScreenWidth} \;;\quad \therefore \quad X_S = X_{ND} \times ScreenWidth$$

$$X_S = X_{ND} \bullet Scale(\, ScreenWidth\,)$$

ND-To-Screen Transform (cont.)

- Similarly for height:



T = ScreenHt – Ys

$$\frac{Y_{ND}}{1} = \frac{T}{ScreenHt} = \frac{(\, ScreenHt - Y_S\,)}{ScreenHt} \;;$$

$$Y_S = (\, Y_{ND} \times (-ScreenHt\,)\,) + ScreenHt$$

$$Y_S = (\, Y_{ND} \bullet Scale\,(\,-ScreenHeight\,)\,) \bullet Translate\,(\,ScreenHeight\,)$$

ND-To-Screen Transform (cont.)

X_S = (X_{ND} • Scale (ScreenWidth)) • Translate (0)

Y_S = (Y_{ND} • Scale (-ScreenHeight)) • Translate (ScreenHeight)

or

P_S = (P_{ND} • Scale (ScreenWidth, -ScreenHeight)) • Translate (0, ScreenHeight)

- In Matrix Form:

$$\begin{pmatrix}X_S\\Y_S\\1\end{pmatrix} = \underbrace{\begin{pmatrix}1&0&0\\0&1&S_{height}\\0&0&1\end{pmatrix}\begin{pmatrix}S_{width}&0&0\\0&-S_{height}&0\\0&0&1\end{pmatrix}}_{\text{ND-to-Screen Transform}}\begin{pmatrix}X_{ND}\\Y_{ND}\\1\end{pmatrix}$$

Combining Transforms

$$\begin{pmatrix}X_S\\Y_S\\1\end{pmatrix} = \underbrace{\begin{pmatrix}\text{ND}\\\text{to}\\\text{Screen}\end{pmatrix} \times \left(\begin{pmatrix}\text{World}\\\text{to}\\\text{ND}\end{pmatrix}\right.}_{} \times \left.\begin{pmatrix}X_W\\Y_W\\1\end{pmatrix}\right)$$

$$\begin{pmatrix}X_S\\Y_S\\1\end{pmatrix} = \begin{pmatrix}\text{“VTM”}\end{pmatrix}\begin{pmatrix}X_W\\Y_W\\1\end{pmatrix}$$

Using The VTM

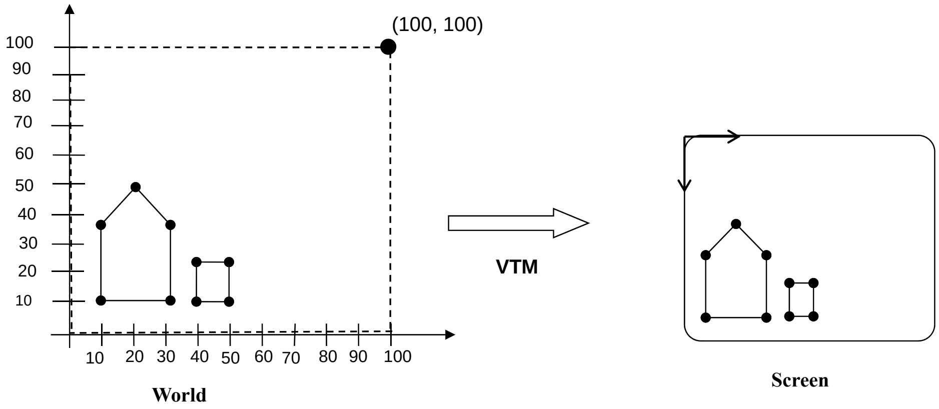
- Suppose we have
  - A panel with access to a collection of *Shapes*
  - Each shape has a **`draw(Graphics2D)`** method which:
    - applies the shape's *transform(s)* to g2d
    - draws the shape in "local" coordinates
    - calls **`draw()`** on any *children*, which applies the child's transforms to g2d and draws the child in "local" coords
- Effect: all draws output *world coordinates*
- We need to *apply the VTM to all output coordinates*

Using The VTM (cont.)

```
public class DisplayPanel extends JPanel {
  // ... code here to provide access to the shape collection for this panel
  AffineTransform worldToND, ndToScreen, theVTM ;
  public void paintComponent (Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D) g ;
    AffineTransform saveAT = g2d.getTransform();
    // update the Viewing Transformation Matrix
    worldToND = buildWorldToNDXform(winWidth,winHeight,winLeft,winBottom);
    ndToScreen = buildNDToScreenXform (panelWidth,panelHeight);
    theVTM = (AffineTransform) ndToScreen.clone();
    theVTM.concatenate(worldToND);          // matrix mult – note order!
    // concatenate the VTM onto the g2d's current transformation
    g2d.transform (theVTM);
    // tell each shape to draw itself using the g2d (which contains the VTM)
    for (Shape s : shapeCollection) {
      s.draw(g2d);
    }
    g2d.setTransform(saveAT);
  }
}
```

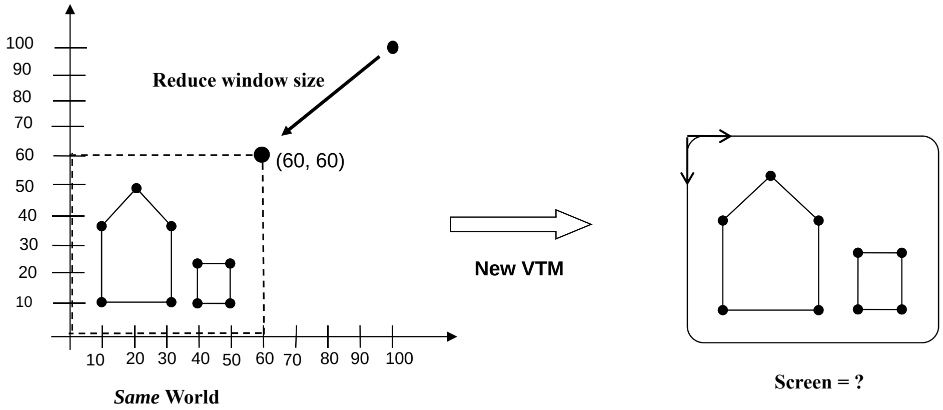
Changing the Window Size

Suppose we start with this:



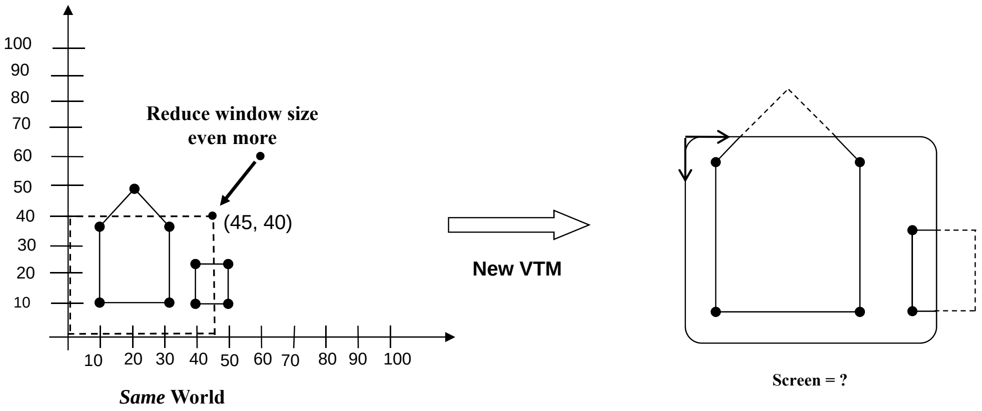
Changing the Window Size (cont.)

Now we change window size,
recompute the VTM, and repaint



Changing the Window Size (cont.)

- Now we change window size *more*, recompute the VTM, and repaint again

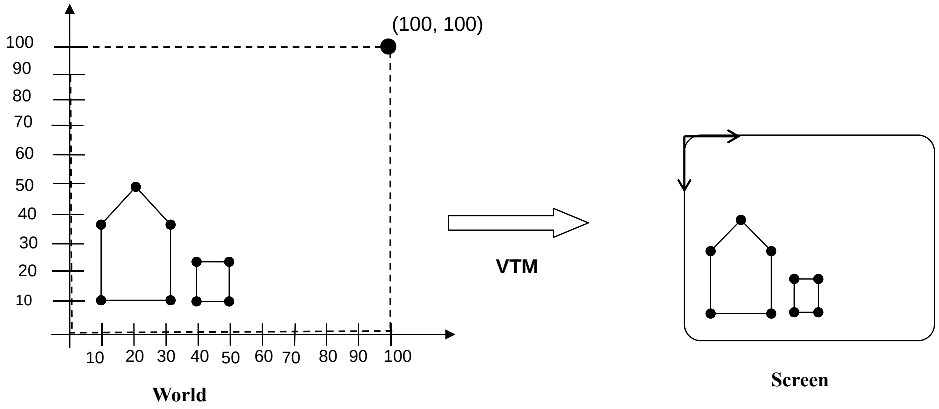


Changing the Window Size (cont.)

```
public class DisplayPanel extends JPanel {

    // world window boundaries
    private double windowLeft, windowRight, windowTop, windowBottom;

    private AffineTransform worldToND, ndToScreen, theVTM;

    public DisplayPanel() {
        //code here to initialize window boundaries and construct the world objects
    }

    //this method changes the window boundaries to be smaller and causes a redraw
    public void zoomIn() {
        double h = windowTop - windowBottom;
        double w = windowRight - windowLeft;
        windowLeft += w*0.05;
        windowRight -= w*0.05;
        windowTop -= h*0.05;
        windowBottom += h*0.05;
        this.repaint();
    }
    //other methods here to perform zoomOut, pan, etc.

    public void paintComponent(Graphics g) {
        //code here to build the VTM, concatenate it onto g2d, and draw the world objects
    }
}
```

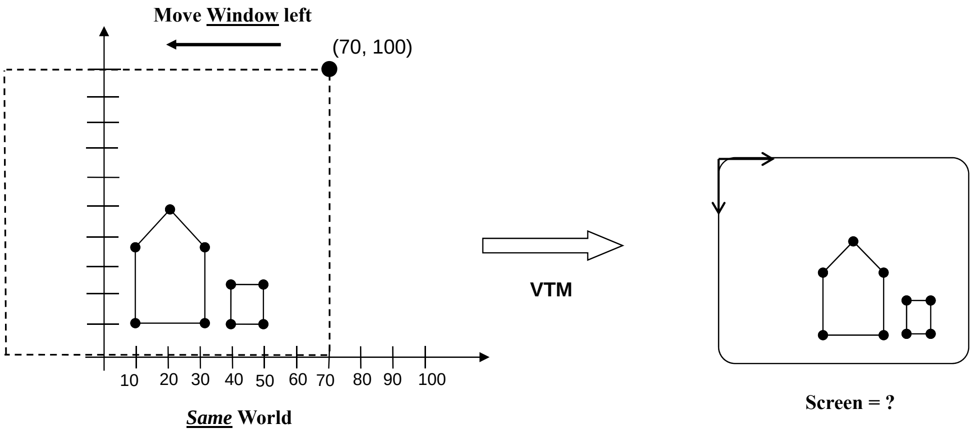
Changing Window *Location*

Suppose we start with this:

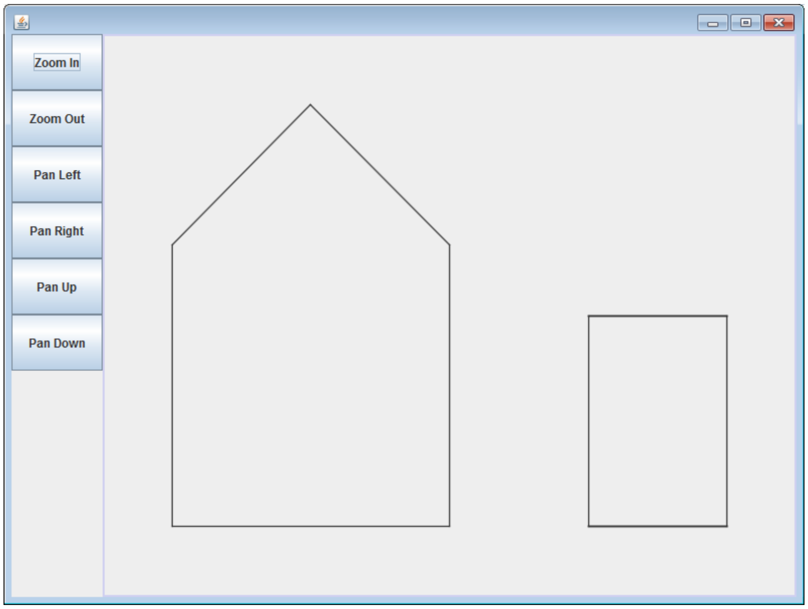


Changing the Window Location (cont.)

Now we change window *location*,
recompute the VTM, and repaint

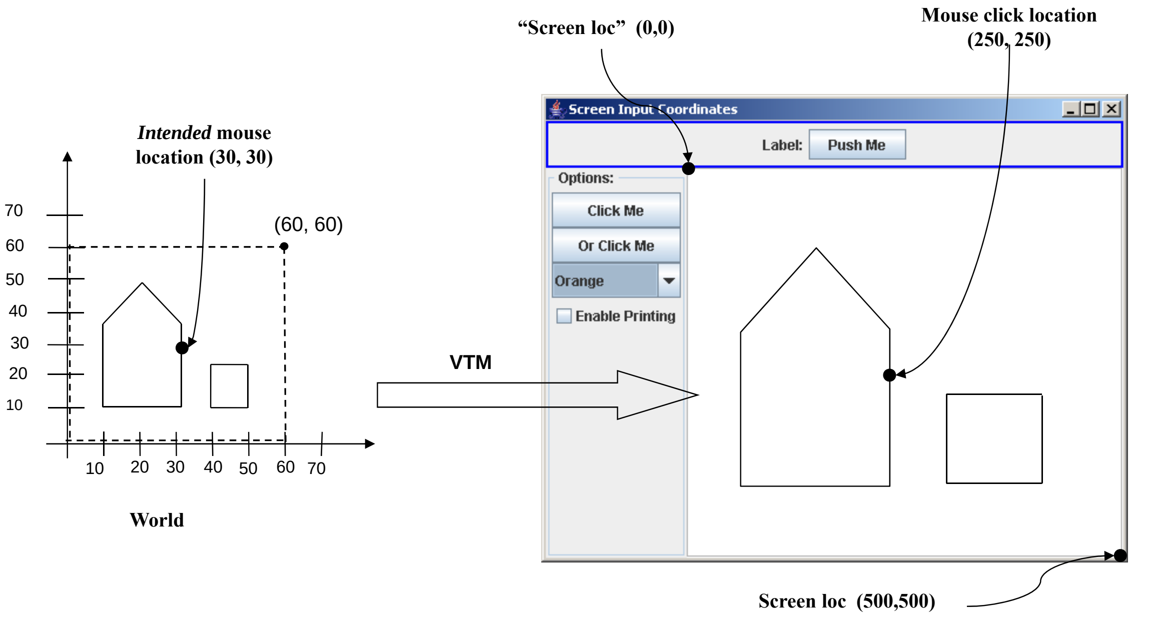


Changing the Window Size (cont.)



Run

Mouse Input: *Screen* Units



Intended mouse location (30, 30)

(60, 60)

World

VTM

"Screen loc" (0,0)

Mouse click location (250, 250)

Screen loc (500,500)

Mapping Mouse To World Coords

When *drawing* (outputting) points, we apply the following transform:

$$\begin{bmatrix} X_s \\ Y_s \\ 1 \end{bmatrix} = \begin{pmatrix} \quad \mathrm{VTM} \quad \end{pmatrix} \times \begin{bmatrix} X_w \\ Y_w \\ 1 \end{bmatrix}$$

Screen Point **World Point**

Mapping To World Coords (cont.)

- **Here, we *have* a *screen* coordinate (the mouse location)**
- **We *need* the corresponding *world* coordinate**
  - We need to go “backwards”:

$$\begin{bmatrix} X_w \\ Y_w \\ 1 \end{bmatrix} = \begin{pmatrix} \quad ?? \quad \end{pmatrix} \times \begin{bmatrix} X_s \\ Y_s \\ 1 \end{bmatrix}$$

World Point **Screen Point (given)**

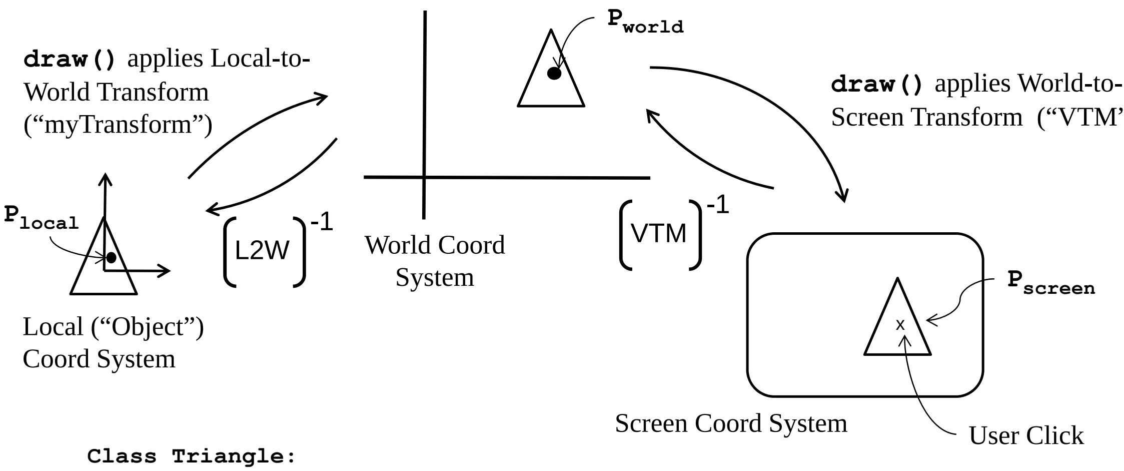
Mapping To World Coords (cont.)

- We need the *inverse* of the VTM

$$\begin{array}{ccccc} \begin{pmatrix} X_w \\ Y_w \\ 1 \end{pmatrix} & = & \begin{pmatrix} \quad \mathbf{VTM} \quad \end{pmatrix}^{-1} & \times & \begin{pmatrix} X_s \\ Y_s \\ 1 \end{pmatrix} \\ \text{World Point} & & & & \text{Screen Point} \end{array}$$

```
AffineTransform theVTM, inverseVTM ;
// ...code here to define the contents of theVTM...
try {
   inverseVTM = theVTM.createInverse();
} catch (NoninvertibleTransformException e) { ... }
Point mouseWorldLoc = inverseVTM.transform(mouseScreenLoc,null);
```

Selection / Containment



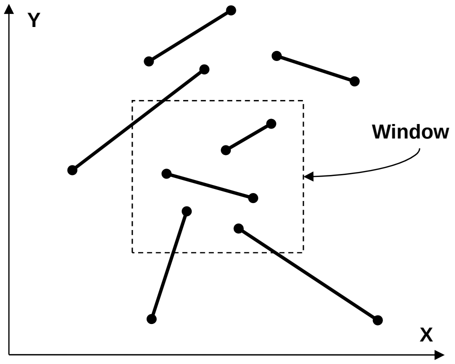
draw() applies Local-to-World Transform ("myTransform")

draw() applies World-to-Screen Transform ("VTM")

```
Class Triangle:
   boolean contains(Point worldPoint) {
      Point localPoint =
            myTransform.createInverse().transform(worldPoint,null);
      return (isInside(localPoint)) ;
   }
```

Clipping

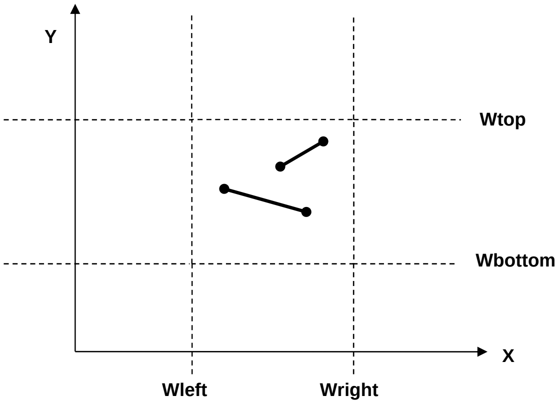
- **Need to suppress output that lies outside the window**
- **For lines, various possibilities:**
  - Both endpoints inside (totally visible)
  - One point inside, the other outside (partially visible)
  - Both endpoints outside (totally invisible ?)



Visibility Tests

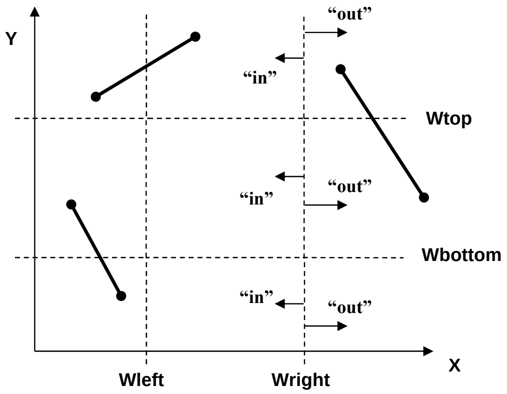
- **"Trivial Acceptance"**
  - Line is completely visible if both endpoints are:

`Below Wtop && Above Wbottom && rightOf Wleft && leftOf Wright`



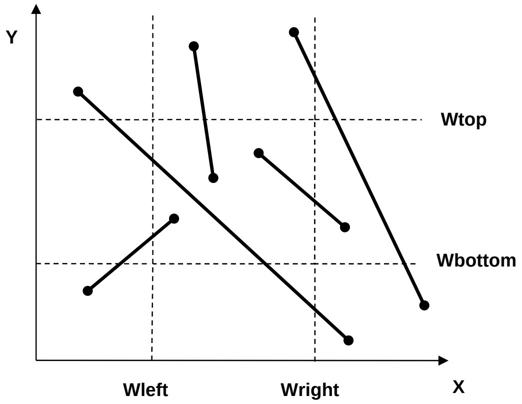
Visibility Tests (cont.)

- **"Trivial Rejection"**
  - Line is completely invisible if both endpoints are on the "out" side of any window boundary



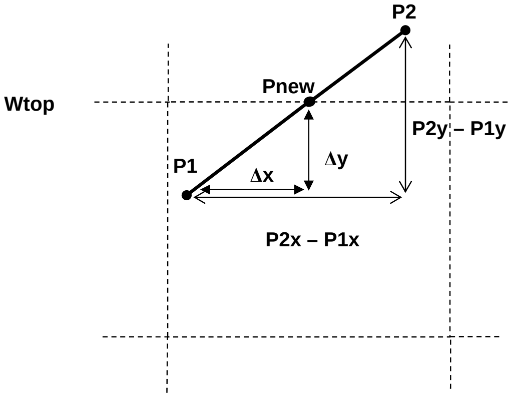
Visibility Tests (cont.)

- Some cases cannot be trivially accepted or rejected :



Clipping Non-Trivial Lines

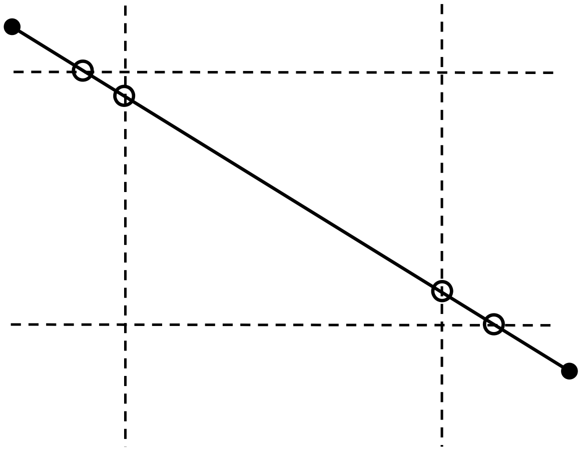
- At least ONE endpoint will be OUTSIDE
  - Compute intersection with (some) boundary
  - Replace "outside" point with Intersection point
  - Repeat as necessary (i.e. until acceptance or empty)



```
Slope = (P2y-P1y) / (P2x-P1x)

PnewY = Wtop

Δy / Δx = Slope

Δy = PnewY – P1y

Δx = Δy / Slope

PnewX = P1x + Δx
```

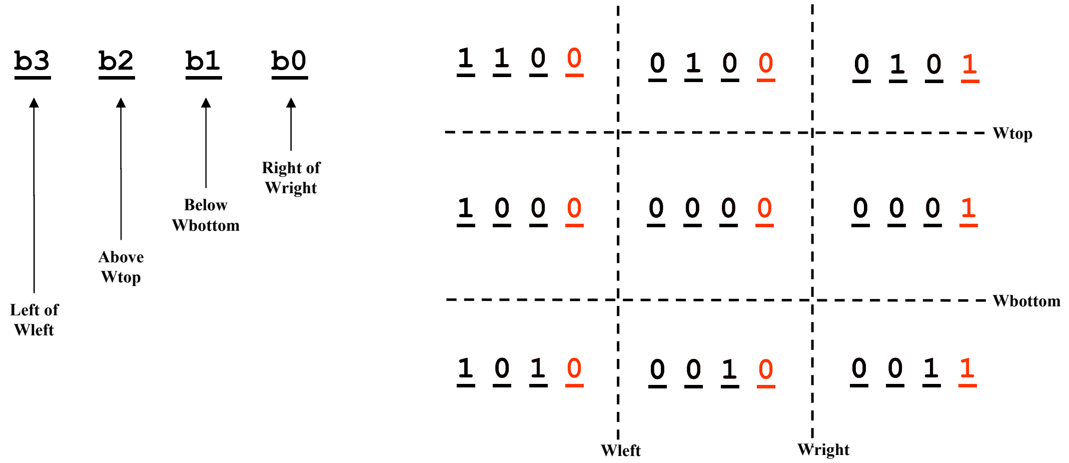
Clipping Non-Trivial Lines (cont.)

- Replacement may have to be done as many as FOUR times:



Cohen-Sutherland Clipping

- Assign ***4-bit codes*** to each Region
  - Each bit-position corresponds to IN/OUT for one boundary



1 1 0 0	0 1 0 0	0 1 0 1
1 0 0 0	0 0 0 0	0 0 0 1
1 0 1 0	0 0 1 0	0 0 1 1

Cohen-Sutherland Clipping (cont.)

- Compare the bit-codes for line end-points
  - Both codes = 0 → trivial acceptance!
    - Center (window) is the only region with code 0000
  - Logical AND of codes != 0 → trivial rejection!

```
code(P1):        b3   b2   b1   b0
code(P2):        b3   b2   b1   b0
               ---------------------
code1 AND code2:  ?    ?    ?    ?
```

← What's required for this to be non-zero?

The Cohen-Sutherland Algorithm

```
/** Clips the line from p1 to p2 against the current world window. Returns the visible
 *  portion of the input line, or returns null if the line is completely outside the window.
 */
Line CSClipper (Point p1,p2) {

   c1 = code(p1); //assign 4-bit CS codes for each input point
   c2 = code(p2);

   // loop until line can be "trivially accepted" as inside the window
   while not (c1==0 and c2==0) {
      // Bitwise-AND codes to check if the line is completely invisible
      if ((c1 & c2) != 0) {
         return null ;     // (logical-AND != 0) means we should reject entire line
      }
      // swap codes so P1 is outside the window if it isn't already
      // (the intersectWithWindow routine assumes p1 is outside)
      if (c1 == 0) {               // if P1 is inside the window
         swap (p1,c1, p2, c2);     // swap points and codes
      }
      // replace P1 (which is outside the window) with a point on the intersection
      // of the line with an (extended) window edge
      p1 = intersectWithWindow (p1, p2);
      c1 = code(p1) ; // assign a new code for the new p1
   }

   return ( new Line(p1,p2) ) ; // the line is now completely inside the window
}
```

The Cohen-Sutherland Algorithm

```
/** Returns a new Point which lies at the intersection of the line p1-p2 with an
 * (extended) window edge boundary line. Assumes p1 is outside the current window.
 */
Point intersectWithWindow (Point p1,p2) {
    if (p1 is above the Window) {
      // find the intersection of line p1-p2 with the window TOP
      x1 = intersectWithTop (p1,p2);         // get the X-intersect
      y1 = windowYMax ;

   } else if (p1 is below the Window) {
      // find the intersection of p1-p2 with the window BOTTOM
      x1 = intersectWithBottom (p1,p2);      // get the X-intersect
      y1 = windowYMin ;

   } else if (p1 is left of the window) {
      // find intersect of p1-p2 with window LEFT side
      x1 = windowXMin  ;
      y1 = intersectWithLeftside (p1,p2)     // get the y-intersect

   } else if (p1 is right of the window) {
      // find intersection with RIGHT side
      x1 = windowXMax ;
      y1 = intersectWithRightside (p1,p2);  // get the y-intersect

   } else {
      return null ; // error – p1 was not outside
   }
   // (x1,y1) is the improved replacement for p1
   return ( new Point(x1,y1) );
}
```

<< This page intentionally left (almost) blank >>

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

XVI – Lines & Curves

- **Rasterization**
- **DDA algorithm**
- **Bresenham's Algorithm**
- **Line-based Graphical Primitives**
- **Parametric Line Representation**
- **Quadratic and Cubic Bezier Curves**
- **Recursive Subdivision**
- **Applications of Curves**

CSc 133 Lecture Notes

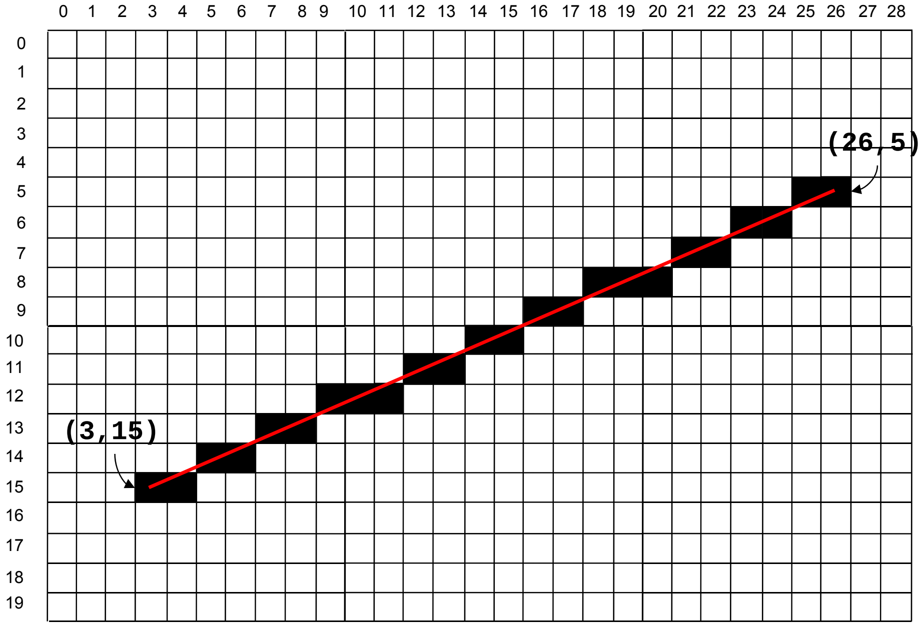
XVI - Lines and Curves

Computer Science Dept.
CSUS

Overview

- **Rasterization**
  - **DDA Algorithm**
  - **Bresenham's Algorithm**
- **Parametric Line Representation**
- **Quadratic & Cubic Bezier Curves**
  - **Geometric and analytical definitions**
- **Rendering Via Recursive Subdivision**
- **Applications of Curves**

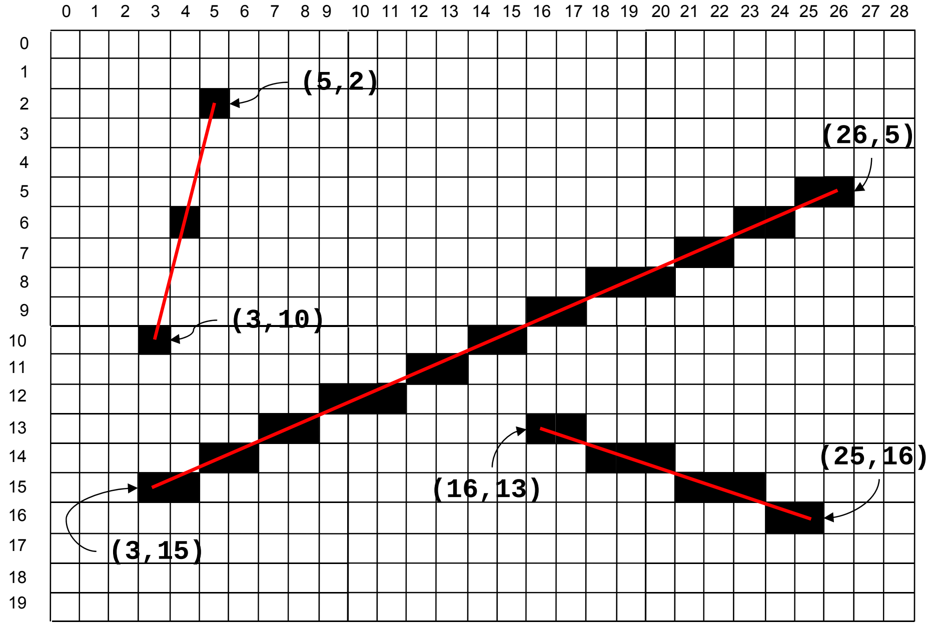
Rasterization



The Simple DDA Algorithm

```
/** Sets pixels on the line between points (xa,ya) and (xb,yb)
 *  to a specified color. This simple version assumes the line
 *  has positive slope < 1.
 */
void simpleLineDDA (int xa,ya, xb,yb; Color rgb) {
  int dx = xb – xa ;           // X-extent of the line
  int dy = yb – ya ;           // Y-extent of the line
  int xIncr = 1 ;              // increase in X per step = 1
  double yIncr = dy/dx ;       // increase in Y per step = slope
  double x = xa ;              // start at first input point
  double y = ya ;
  setPixel ((int)x, (int)y, rgb) ;
  for (int k=1; k<=dx; k++) {
    x = x + xIncr ;
    y = y + yIncr ;
    setPixel (round(x), round(y), rgb) ;
    }
}
```

Applying The DDA Algorithm

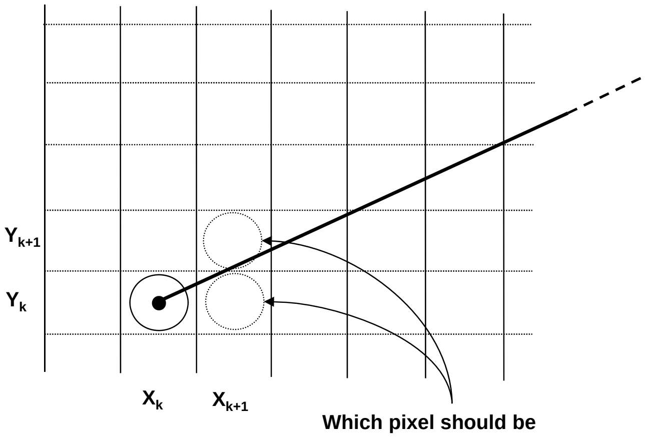


Full DDA Algorithm

```
/** Sets pixels on the line between points (xa,ya) and (xb,yb) to a specified color.
 *  Works for lines of arbitrary slope with positive or negative direction.
 */
void LineDDA (int xa,ya, xb,yb; Color rgb) {
  int dx, dy ;              // distance in X and Y for the line
  int steps ;               // number of unit steps in X or Y
  double x, y ;             // ’current’ loc on the line
  double xIncr, yIncr ;     // increment per step in X and Y
  dx = xb – xa ;            // X-extent of the line
  dy = yb – ya ;            // Y-extent of the line
  if abs(dx) > abs (dy)
  then
    steps = abs (dx)        // slope < 1, so unit steps in X
  else
    steps = abs (dy) ;      // slope >= 1; unit steps in Y
  xIncr = dx / steps ;      // increase in X per step
  yIncr = dy / steps ;      // increase in Y per step
  x = xa ;                  // start at first input point
  y = ya ;
  setPixel ((int)x, (int)y, rgb) ;
  for (int k=1; k<=steps; k++) {
    x = x + xIncr ;
    y = y + yIncr ;
    setPixel (round(x), round(y), rgb) ;
  }
}
```

The “Pixel Selection” Decision

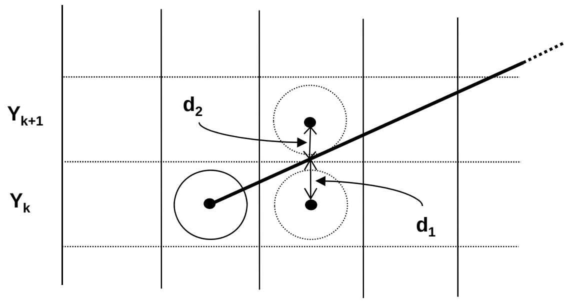
- Basic question: which is the best “next pixel”?



Which pixel should be selected at X_{k+1} ?

The “Pixel Decision” Parameter

- Choose the pixel *closest to the true line*



```
if ((d1-d2) > 0)
   choose pixel $Y_{k+1}$
else
   choose pixel $Y_k$
```

Same as “sign(d1-d2) is +”

Bresenham’s Algorithm

```
/** Integer-based scan-conversion for lines with positive slope < 1  */
void simpleBresenham (int xStart, yStart, xEnd, yEnd; Color rgb) {
  int x = xStart ;
  int y = yStart ;
  int deltaX = xEnd – xStart ;
  int deltaY = yEnd – yStart ;
  int p = 2 * deltaY – deltaX ;    // ’decision parameter’ p0 for the initial point
  setPixel (x,y,rgb) ;             // plot the first point
  while (x < xEnd) {
     x = x + 1 ;                   // unit increase in X since slope < 1
     if (p < 0) {
        setPixel (x, y, rgb) ;     // p<0 implies Y should not increase
        p = p + 2*deltaY ;         // new p(k+1) for next iteration
     }
     else {
        y = y + 1 ;                // p>=0 implies moving up one in Y
        setPixel (x, y, rgb) ;
        p = p + 2*deltaY – 2*deltaX ;
     }
  }
}
```

The Math Behind Bresenham

For lines with a positive slope < 1, given a previously plotted point (X_k, Y_k), we want to determine whether the Y value at X_{k+1} should be Y_k (i.e. continue horizontally with the next pixel), or Y_{k+1} (i.e. move up one pixel in Y as we move right one pixel in X). This is determined by whether the true value Y of the line, calculated at X_{k+1}, is closer to Y_k or Y_{k+1}.

We call the distance from pixel (X_{k+1}, Y_k) to Y (the true value of the line) “d1”; and the distance from Y to pixel (X_{k+1}, Y_{k+1}) “d2”. Then, if d1 < d2, the next pixel should be (X_{k+1},Y_k) (that is, the next Y pixel is on the same scan line as the previous one); but if d1 >= d2, the next pixel should be (X_{k+1}, Y_{k+1}) (that is, the Y pixel moves up one scan line).

Said another way, if the difference (d1 – d2) is negative, d2 is greater than d1 and we should plot (X_{k+1}, Y_k); but if (d1 – d2) is non-negative then d1 is greater and the true value of the line lies closer to pixel (X_{k+1}, Y_{k+1}).

We have the following formulae:

$y = m\,(X_{k+1}) + b$ // the true Y value of the line at the next pixel

$d1 = y – Y_k$ // the distance between the current Y pixel and the true Y value

$= [m\,(X_{k+1}) + b] - Y_k$

$d2 = Y_{k+1} - y$ // the distance between the true Y value and the next higher pixel

$= Y_{k+1} - [m\,(X_{k+1}) + b]$

$= Y_{k+1} - m(X_{k+1}) - b$

$d1 – d2 = ([m\,(X_{k+1}) + b] - Y_k) - (Y_{k+1} - m(X_{k+1}) - b)$ // the “decision parameter”

$= (2m(X_{k+1}) - 2Y_k + 2b - 1)$

The Math Behind Bresenham (cont.)

Now, consider a newly-defined "parameter" term

$p = \Delta x\ (d1 - d2)$ // Δx = change in X of slope

Because we are assuming a positive slope < 1, the sign of p will be the same as the sign of (d1 – d2).

$p = \Delta x\ (d1 - d2)$

$= \Delta x\ (2m(X_{k+1}) - 2Y_k + 2b - 1)$

$= \Delta x\ (2\ (\Delta y/\Delta x)\ (X_{k+1}) - 2Y_k + 2b - 1)$

$= 2\Delta y(X_{k+1}) - 2\Delta x\ Y_k + \Delta x\ (2b - 1)$

But since we are stepping in unit increments in the X direction,

$X_{k+1} = X_k + 1$, so we can rewrite the above as

$p = 2\Delta y(X_k + 1) - 2\Delta x\ Y_k + \Delta x\ (2b - 1)$

$= 2\Delta y(X_k) - 2\Delta x\ Y_k + C$, where $C = \Delta x\ (2b - 1) + 2\Delta y$

Thus for any X value X_k, we can determine its "parameter" value p; and further, the sign of p indicates whether the Y value at X_k steps up to a new scan line from the previous X location or not.

The Math Behind Bresenham (cont.)

Next, consider the "parameter value" P_{k+1}:

$P_{k+1} = 2\Delta y(X_{k+1}) - 2\Delta x(Y_{k+1}) + C$

And consider the difference $(P_{k+1} - P_k)$:

$P_{k+1} - P_k = (2\Delta y(X_{k+1}) - 2\Delta x(Y_{k+1}) + C) - (2\Delta y(X_k) - 2\Delta x\ (Y_k) + C)$

But since we are stepping in unit increments in X, substituting $(X_k + 1)$ for X_{k+1} (as above) we get

$P_{k+1} - P_k = (2\Delta y(X_k + 1) - 2\Delta x(Y_{k+1}) + C) - (2\Delta y(X_k) - 2\Delta x\ (Y_k) + C)$

$P_{k+1} - P_k = 2\Delta y(\ X_k + 1 - X_k) - 2\Delta x(Y_{k+1} - Y_k) + C - C$

And therefore since $(X_k + 1 - X_k) = 1$,

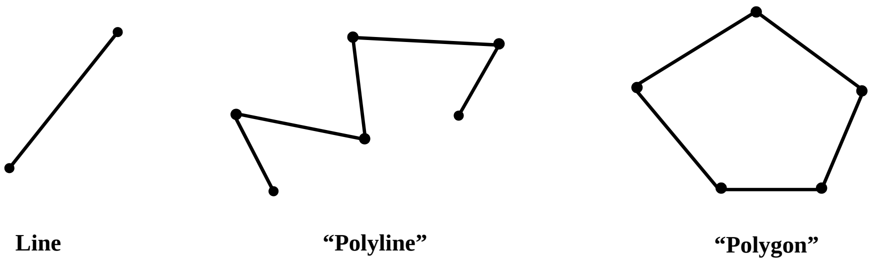
$P_{k+1} = P_k + 2\Delta y - 2\Delta x(Y_{k+1} - Y_k)$

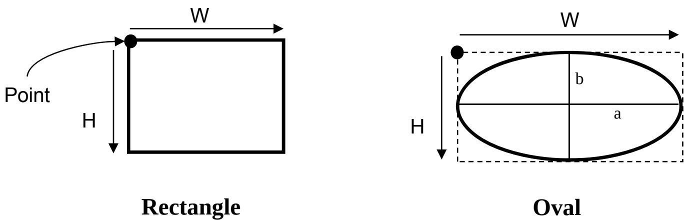
Thus we have an integer formula for computing each P_{k+1} from the previous P_k. Further, if the sign of P_k is the same as the sign of (d1 – d2) (that is, if $P_k < 0$), then the last term in the preceding equation is zero (because Y_{k+1} is the same pixel as Y), and only the first two terms in the preceding equation are needed to compute P_k; otherwise, $Y_{k+1} - Y_k = 1$, and all three terms are used to calculate P_{k+1} from P_k.

Thus the Bresenham algorithm involves two basic steps: (1) for each pixel X_k, use the sign of the corresponding value P_k to determine whether the Y pixel at K should be the same as the previous Y, or should be Y_{k+1}; and (2) use P_k and the constant $2\Delta y$ (and the constant $2\Delta x$ if $P_k < 0$) to calculate P_{k+1}.

Graphical Primitives

- Point- and Line-based

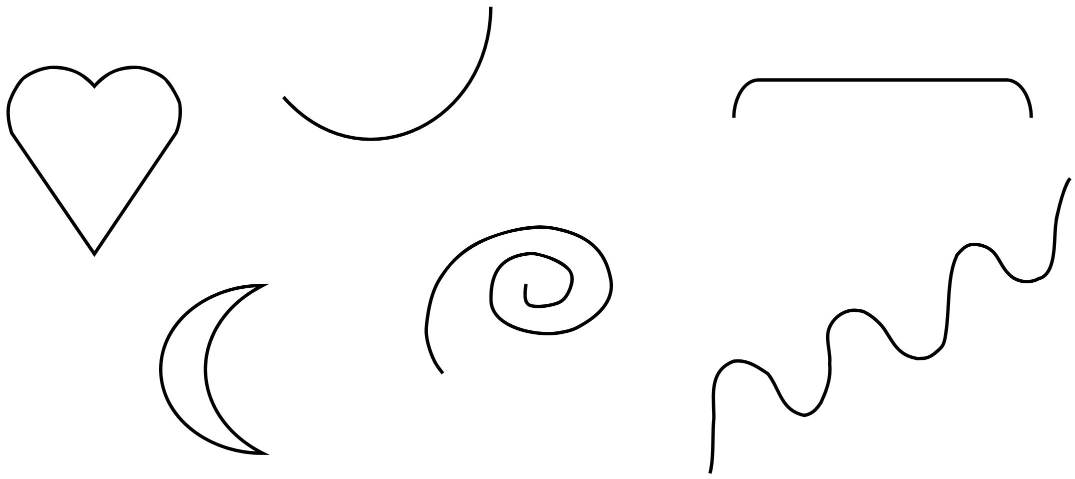




$$\frac{(x - xCenter)^2}{a^2} + \frac{(y - yCenter)^2}{b^2} = 1$$

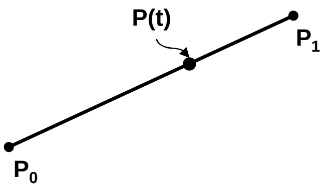
Curves Of Higher Complexity

- What if we want to draw shapes like these?



Parametric Line Representation

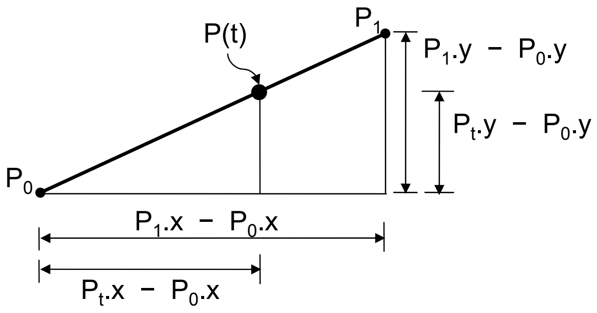
- Lines can be represented in terms of known quantities in several ways :
  - `Y = mX + b`// line with slope “m” and Y-intercept “b”
  - `(P0, P1)` // line containing $\mathbf{P_0}$ and $\mathbf{P_1}$
- Any point on **(P_0, P_1)** can be represented with a single *parameter value ‘**t**’*



- **‘t’ is the ratio of [distance from P_0 to P(t)] to [distance from P_0 to P_1]**
- **Every point on the line has a unique ‘t’ value**

Parametric Line Representation (cont.)

- Parametric equation for points ***P(t)*** on a line:



$$t = \frac{P_t - P_0}{P_1 - P_0}$$

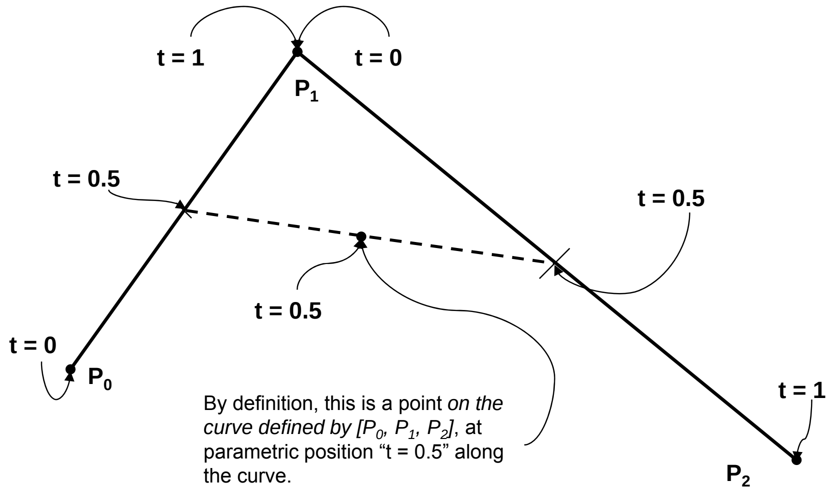
$$t\left(P_1 - P_0\right) = P_t - P_0$$

$$P_t = P_0 + t\left(P_1 - P_0\right)$$

$$P_t = \left(1 - t\right) P_0 + t\, P_1$$

Quadratic Bezier Curves

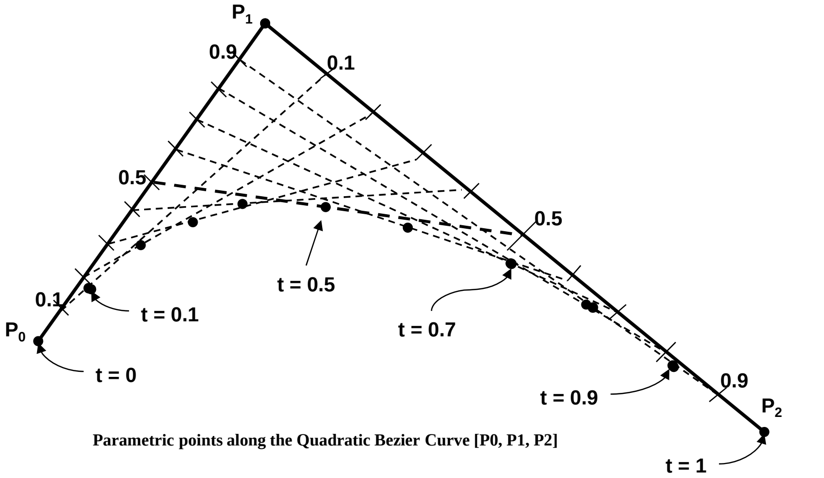
- Geometric description



By definition, this is a point *on the curve defined by [P_0, P_1, P_2]*, at parametric position "t = 0.5" along the curve.

Quadratic Bezier Curves (cont.)

- Connecting points of equal parametric value generates a curve:

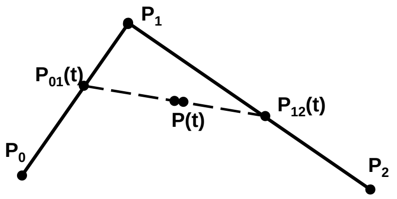


Parametric points along the Quadratic Bezier Curve [P0, P1, P2]

Demo

Quadratic Bezier Curves (cont.)

- Analytic definition



$$P_{01}(t) = t \cdot P_1 + (1-t) \cdot P_0 \qquad [1]$$

and

$$P_{12}(t) = t \cdot P_2 + (1-t) \cdot P_1 \qquad [2]$$

and a point on the curve $[P_0 \quad P_1 \quad P_2]$ is defined as

$$P(t) = t \cdot (P_{12}(t)) + (1-t) \cdot (P_{01}(t)) \qquad [3]$$

Substituting [1] and [2] into [3] gives

$$P(t) = t \cdot (t \cdot P_2 + (1-t) \cdot P_1) + (1-t) \cdot (t \cdot P_1 + (1-t) \cdot P_0)$$

Factoring and combining the constant terms P_0, P_1, and P_2 gives

$$P(t) = (1-t)^2 \cdot P_0 + \left(-2t^2 + 2t\right) \cdot P_1 + \left(t^2\right) \cdot P_2$$

Curves as Weighted Sums

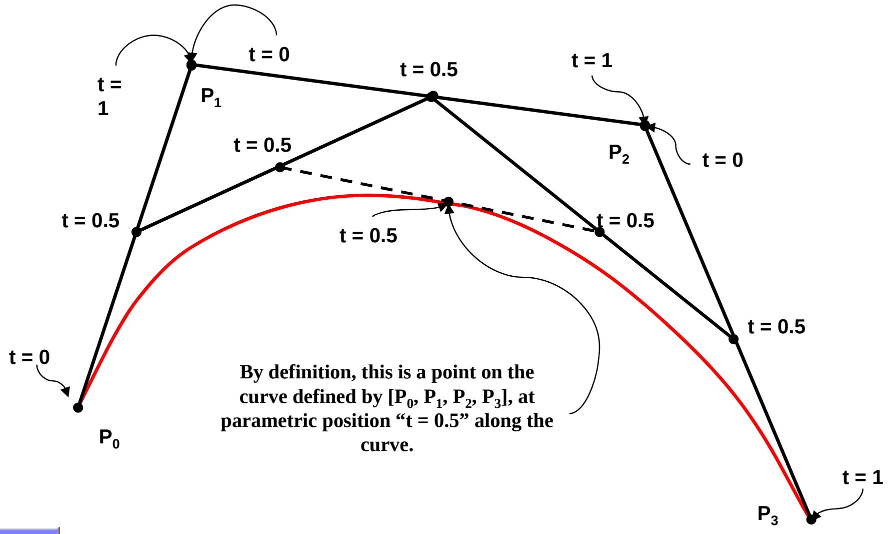
$$P(t) = (1-t)^2 \cdot P_0 + \left(-2t^2 + 2t\right) \cdot P_1 + \left(t^2\right) \cdot P_2$$

$$P(t) = \sum_{i=0}^{2} P_i \cdot B_i(t), where \begin{cases} B_0(t) = (1-t)^2 \\ B_1(t) = \left(-2t^2 + 2t\right) \\ B_2(t) = t^2 \end{cases}$$

- *A point on the curve is a <u>weighted sum</u> of the three "control points"*
  - The "weightings" are the quadratic polynomials, evaluated at "*t*"

Cubic Bezier Curves

- Geometric description

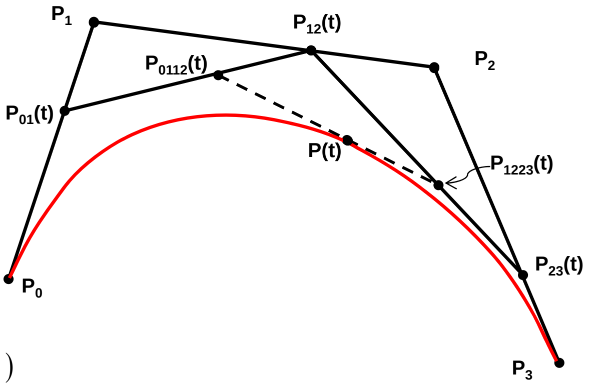


By definition, this is a point on the curve defined by [P_0, P_1, P_2, P_3], at parametric position “t = 0.5” along the curve.

Demo

Cubic Bezier Curves (cont.)

- Analytic definition



$$P_{01}(t) = t\cdot P_1 + (1-t)\cdot P_0$$

$$P_{12}(t) = t\cdot P_2 + (1-t)\cdot P_1$$

$$P_{23}(t) = t\cdot P_3 + (1-t)\cdot P_2$$

$$P_{0112}(t) = t\cdot P_{12}(t) + (1-t)\cdot P_{01}(t)$$

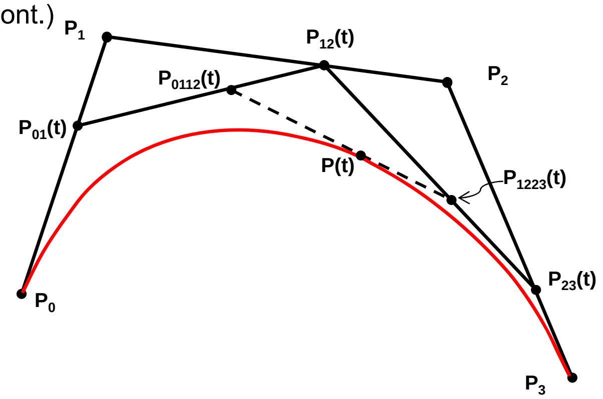
$$P_{1223}(t) = t\cdot P_{23}(t) + (1-t)\cdot P_{12}(t)$$

and a point on the curve $[\,P_0 \quad P_1 \quad P_2 \quad P_3\,]$ is defined as

$$P(t) = t\cdot(P_{1223}(t)) + (1-t)\cdot(P_{0112}(t))$$

Cubic Bezier Curves (cont.)

- Analytic definition (cont.)



$$
\begin{aligned}
P(t) &= t\cdot\left(P_{1223}(t)\right) \quad + \quad (1-t)\cdot\left(P_{0112}(t)\right)\\
&= \underline{(1-t)^3}\cdot P_0 \quad + \quad \underline{\left(3t^3-6t^2+3t\right)}\cdot P_1 \quad + \quad \underline{\left(-3t^3+3t^2\right)}\cdot P_2 \quad + \quad \underline{\left(t^3\right)}\cdot P_3\\
&= \sum_{i=0}^{3} P_i \cdot B_{i,3}(t)
\end{aligned}
$$

Drawing Bezier Curves

- Iterative approach

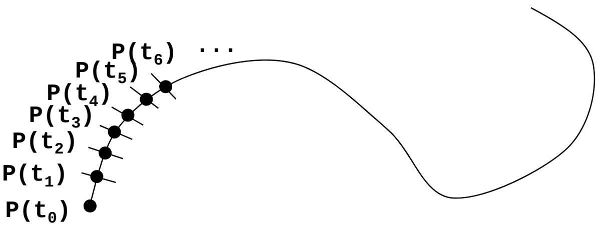
moveTo (P(t_0));

drawTo (P(t_1));

drawTo (P(t_2));

drawTo (P(t_3));

...



Drawing Bezier Curves (cont.)

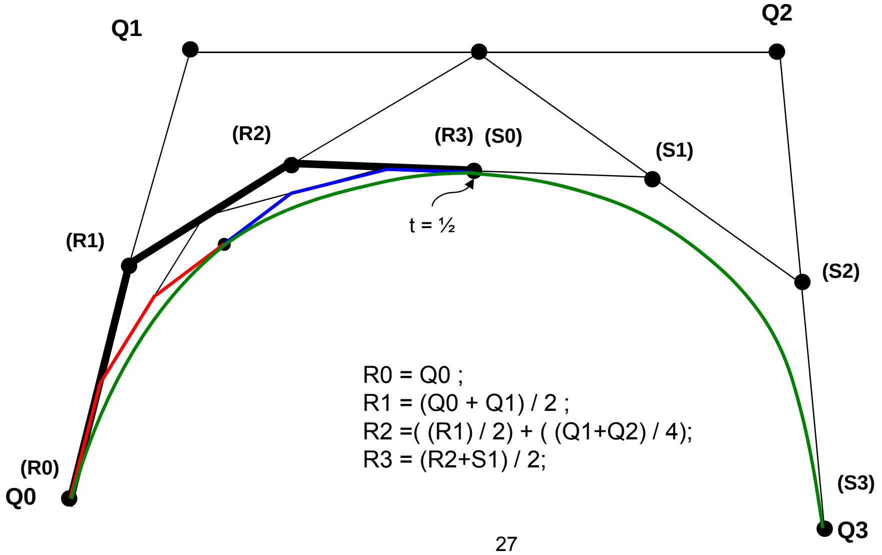
```
/** A routine to draw the (cubic) Bezier Curve represented by the (1x4) input
 *  Control Point Array using iterative plotting along the curve and an explicit
 *  computation which produces a weighted sum of control points for each new point.
 */
void drawBezierCurve (controlPointArray) {
  currentPoint = controlPointArray [0] ; // start drawing at first control point
  t = 0 ;   // vary the parametric value "t" over the length of the curve
  while ( t<=1 ) {
    // compute next point on the curve as the sum of the Control Points, each
    // weighted by the appropriate polynomial evaluated at ‘t’.
    nextPoint = (0,0) ;
    for (int i=0; i<=3; i++) {
      nextPoint = nextPoint + ( blendingFunction(i,t) * controlPointArray[i] );
    }
    drawLine (currentPoint,nextPoint);
    currentPoint = nextPoint;
    t = t + smallFloatIncrement;
  }
}
```

Drawing Bezier Curves (cont.)

```
/** Returns the value of the “ith” cubic Bernstein polynomial blending
 *  function at parametric location ‘t’
 */
double blendingFunction (int i, double t) {
  switch (i)  {
     case 0: return ( (1-t) * (1-t) * (1-t) ) ;     // (1-t)^3
     case 1: return ( 3 * t * (1-t) * (1-t) ) ;     // 3t(1-t)^2
     case 2: return ( 3 * t * t * (1-t) ) ;         // 3t^2(1-t)
     case 3: return ( t * t * t ) ;                 // t^3
  }
}
```

Control Mesh Subdivision

- Split the control mesh [Q] at t=1/2
  - Produces two meshes [R] and [S]



R0 = Q0 ;
R1 = (Q0 + Q1) / 2 ;
R2 =((R1) / 2) + ((Q1+Q2) / 4);
R3 = (R2+S1) / 2;

Recursive Subdivision

```
/** Draws the (cubic) Bezier curve represented by the (1x4) input Control Point Vector
 *  by recursively subdividing the Control Point Vector until the control points are
 *  within some tolerance of being colinear, at which time the Control Points are deemed
 *  "close enough" to the curve for the 1st and last control points to be used as the
 *  ends of a line segment representing a short piece of the actual Bezier curve. */
void drawBezierCurve (ControlPointVector, Level) {
  if ( straightEnough (ControlPointVector) OR (Level>MaxLevel) )
      Draw Line from 1st Control Point to last Control Point ;
  else  {
      subdivideCurve (ControlPointVector, LeftSubVector, RightSubVector) ;
      drawBezierCurve (LeftSubVector,Level+1) ;
      drawBezierCurve (RightSubVector,Level+1) ;
  }
}
 /** Splits the input control point vector Q into two control point
  *  vectors R and S such that R and S define two Bezier curve segments that
  *  together exactly match the Bezier curve defined by Q.
  */
void subdivideCurve (ControlPointVector Q,R,S) {
  R(0) = Q(0) ;
  R(1) = (Q(0)+Q(1)) / 2.0 ;
  R(2) = (R(1)/2.0) + (Q(1)+Q(2))/4.0 ;
  S(3) = Q(3) ;
  S(2) = (Q(2)+Q(3)) / 2.0 ;
  S(1) = (Q(1)+Q(2))/4.0 + S(2)/2.0 ;
  R(3) = (R(2)+S(1)) / 2.0 ;
  S(0) = R(3) ;
}
```

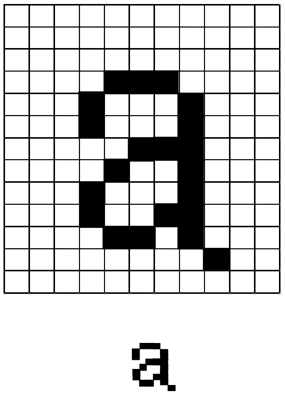
Recursive Subdivision (cont.)

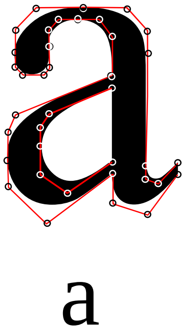
```
/** determines whether the four points Q0,Q1,Q2,Q3 in the input array of Control
 * Points are within some tolerance "epsilon" of being colinear.
 */
boolean straightEnough (ControlPointVector) {
  // find length around control polygon
  d1 = lengthOf(Q0,Q1) + lengthOf(Q1,Q2) + lengthOf(Q2,Q3);
  // find distance directly between first and last control point
  d2 = lengthOf(Q0,Q3) ;
  if ( abs(d1-d2) < epsilon )      // epsilon (“tolerance”) = (e.g.) .001
      return true ;
  else
      return false ;
}
```

Demo

Applications Of Curves

- Two types of “fonts”
  - Bit-mapped
  - Outline





<< This page intentionally left (almost) blank >>

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

XVII – Component Models

- **The Façade design pattern**
- **Components**
- **JavaBeans, EJBs, COM and DCOM**
- **An Overview of Reflection (Introspection)**
- **The Java Reflection classes**
- **GUI Builders**

CSc 133 Lecture Notes

XVII - Component Models

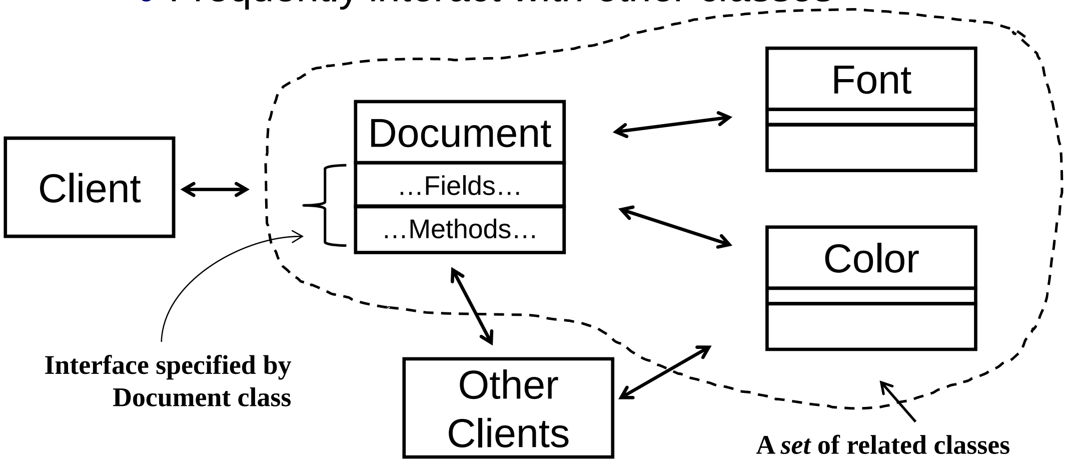
Computer Science Dept.
CSUS

Overview

- Class Interactions
- The Façade Design Pattern
- Components
  - Java Beans, EJBs, COM & DCOM
- Reflection
- Component-Based Tools
  - GUI Builder Example

Class Interactions

- Classes are *encapsulations of abstractions*
  - Contain collections of *fields* and *methods*
  - Define an *interface* accessible to client
  - Frequently *interact with other classes*



Client

Document

...Fields...

...Methods...

Font

Color

Other Clients

Interface specified by Document class

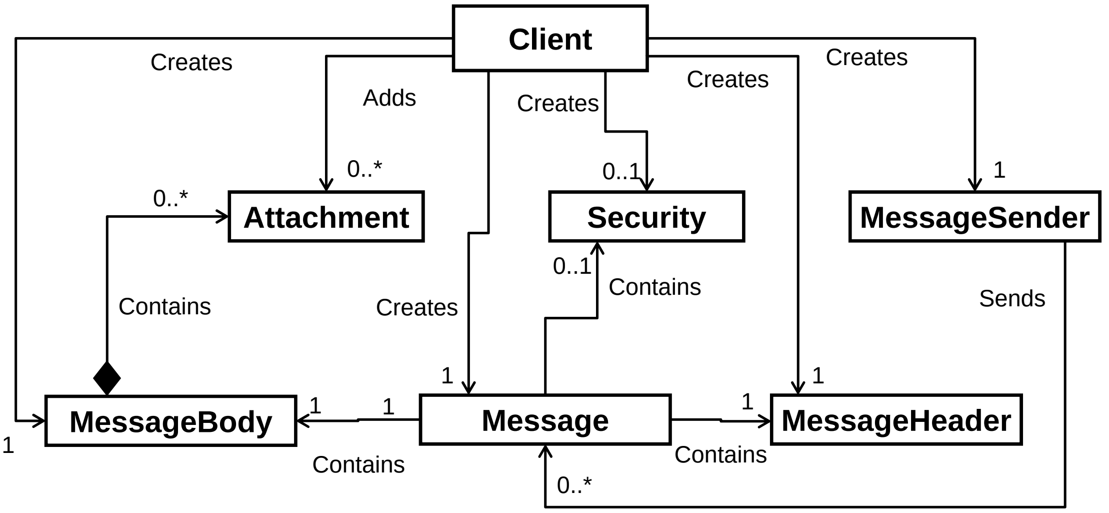
A *set* of related classes

Example: Interacting Classes

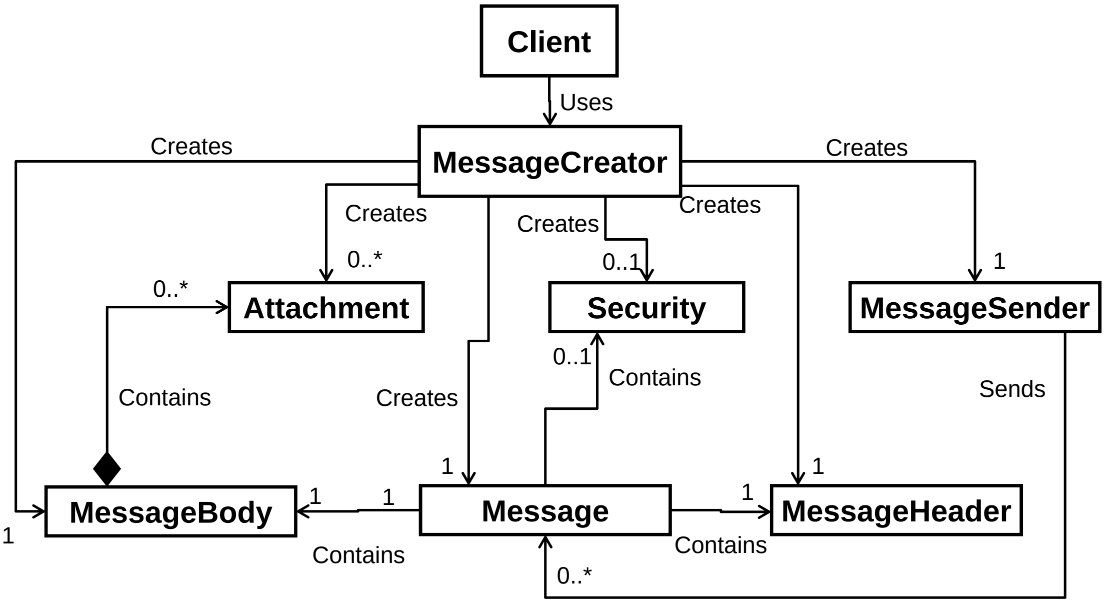
- Consider a set of classes supporting creation/sending of an email message:
  - Class **`MessageBody`**
  - Class **`Attachment`**
  - Class **`MessageHeader`**
  - Class **`Message`** (contains **`Body`**, **`Attachments`**, **`Header`**)
  - Class **`Security`** (used to add digital signature)
  - Class **`MessageSender`** (sends Messages)

After an example in Patterns in Java, Vol.1, M. Grand

Client Use Of Email Classes



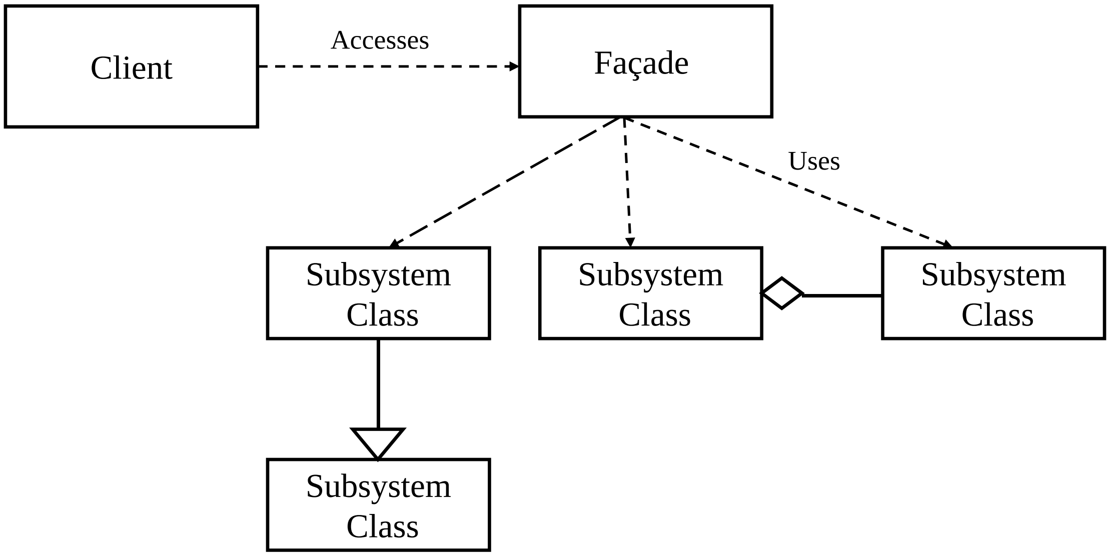
Simplifying Interactions



FAÇADE Design Pattern

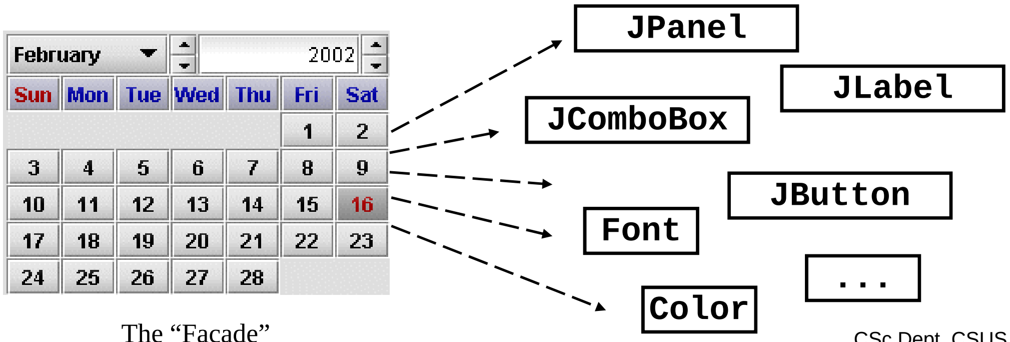
- Provides a unified interface to a set of interfaces in a subsystem.
- Defines a higher-level interface that makes the subsystem easier to user
  - Provides a coherent entry point to the capabilities of the subsystem.
  - Subsystem implementation is subject to change, but ***the functionality that it provides is stable***.

UML for Façade



Components

- A building block that can be combined with other components into programs
  - Bigger unit than a normal object (generally higher level than program classes)
  - Intended to be composed with other code (re-used)
  - Only minimal additional programming for reuse
- Example: Calendar component



February ▾ 2002

Sun	Mon	Tue	Wed	Thu	Fri	Sat
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28		

JPanel

JLabel

JComboBox

JButton

Font

...

Color

The “Façade”

Java Beans

- Java’s *component model*
  - Well-defined (specified) interface
  - Composed of one or more classes
- Typical Capabilities
  - Execute methods
  - Expose properties
  - Emit events
  - Save state (“persistence”)

JavaBean Attributes

- **Minimum Requirements:**
  - Serializable
  - Public no-argument constructor
- **Most Beans also have**:
  - Properties
    - Public getters/setters must follow naming convention
    - Can be *single* (e.g . **`String name = bean.getName()`**) or *indexed* (e.g. **`int [] scores = bean.getScores()`**)
    - Can be *bound*, meaning changes invoke *PropertyChangeListeners*
    - Can be *constrained*, meaning changes are limited by consulting *VetoListeners*
  - Methods
    - Allow client code to manipulate bean state beyond just property changes
  - Events
    - Beans can fire events, including user-defined events
    - Clients can add/remove event listeners to beans

Example: FaceBean

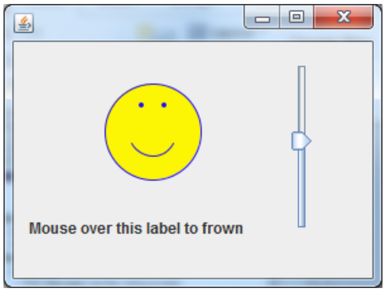
```
//Defines a JavaBean which displays a face with a smile (or frown) with variable width.
// Based on an example from http://docs.oracle.com/javase/tutorial/javabeans
public class FaceBean extends JComponent {
    //Bean properties
    private int mouthWidth = 90;
    private boolean smile = true;
    public void paintComponent(Graphics g) {
        //code here to draw a face with a mouth whose size depends on the current
        // ‘mouthWidth’ property and whose orientation depends on current ‘smile’ property
    }
    //Bean property accessors
    public int getMouthWidth() {
        return mouthWidth;
    }
    public void setMouthWidth(int mw) {
        mouthWidth = mw;
        repaint();
    }
    //Bean methods
    public void smile() {
        smile = true;
        repaint();
    }
    public void frown() {
        smile = false;
        repaint();
    }
}
```

Example 2: FacePanel Bean

```
//Defines a panel which is a JavaBean and which contains a FaceBean along with a
// mouse-sensitive Label and a Slider which manipulates the FaceBean’s mouth width
public class FacePanelBean extends JPanel implements ChangeListener {
    private FaceBean faceBean;
    private JLabel label;
    private JSlider slider;
    public FacePanelBean() {
        label = new JLabel();
        slider = new JSlider();
        slider.addChangeListener(this);   //the panel bean listens for slider changes
        faceBean = new FaceBean();
        label.setText("Mouse over this label to frown");
        label.addMouseListener(new MouseAdapter() {
            public void mouseEntered(MouseEvent e) {
                faceBean.frown();
            }
            public void mouseExited(MouseEvent e) {
                faceBean1.smile();
            }
        });
        //code here to set the slider properties (range and orientation), the FaceBean
        // color properties, and the panel layout
    }
    //invoked whenever the slider changes state
    public void stateChanged(ChangeEvent e) {
        faceBean.setMouthWidth(slider.getValue());
    }
}
```

Using the FacePanel Bean

```
//Defines a frame which displays a FacePanelBean
public class FaceFrame extends JFrame {

    public FaceFrame() {
        FacePanelBean fp = new FacePanelBean();
        this.add (fp, BorderLayout.CENTER);
    }

    public static void main(String[] args) {
        FaceFrame f = new FaceFrame();
        f.setSize(300,225);
        f.setVisible(true);
    }

}
```



Run

Reflection

- The ability of a program to analyze its objects and their capabilities
  - Also called “introspection”
- Supported in Java via classes that describe classes (and their parts)
  - Generally referred to as the “Reflection Classes”

The Java Reflection Classes

Class Name	Purpose
Class	Describes the class of an object
Package	Describes a package
Field	Describes a field and allows inspection and modification of fields
Method	Describes a method and allows its invocation on objects
Constructor	Describes a constructor and allows its invocation
Array	Has static methods to analyze arrays

The **`Class`** Class

- Every object contains a “**`Class`**” object
- The **`Class`** object *describes the class of the object containing it*
- Example methods:
  - **`getSuperClass()`**
  - **`getInterfaces()`**
  - **`getDeclaredFields()`**
  - **`getDeclaredMethods()`**

Example: Tank Class

```
public class Tank extends WorldObject
                implements IMovable
{
  private Color myColor;

  public Tank(Color theColor)
  { myColor = theColor; }

  public void move(int deltaX, int deltaY)
  {
   int moveAmount = smallerOf(deltaX, deltaY);
   setX ( getX() + moveAmount) ;
   setY ( getY() + moveAmount) ;
  }

  private int smallerOf (int arg1, int arg2)
  {
   if (arg1<=arg2){
      return arg1;
   } else {
      return arg2;
   }
  }
}
```

```
public interface IMovable
{
  public void move(int x,int y);
}
```

```
public class WorldObject
{
   private int x,y;
   protected int getX()
   { return x; }

   protected int getY()
   { return y; }

   protected void setX(int newX)
   { x = newX; }

   protected void setY(int newY)
   { y = newY; }
}
```

Example: Using the `Class` Class

```
import java.awt.Color;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class TankTest {
  public static void main (String [] args) {
     Tank myTank = new Tank(Color.red);
     Class tankClass = myTank.getClass();
     System.out.println ("Tank's superclass = "+tankClass.getSuperclass());
     Field [] fields = tankClass.getDeclaredFields() ;
     System.out.println ("Tank declares fields: " );
     for (Field f : fields)                {
        System.out.println (" '" + f + "'");
     }
     System.out.println ("");

     Method [] methods = tankClass.getDeclaredMethods() ;
     System.out.println ("Tank declares methods: " );
     for (Method m : methods) {
        System.out.println (" '" + m + "'");
     }
     System.out.println ("");
  }
}
```

Run

Example 2: Instantiation

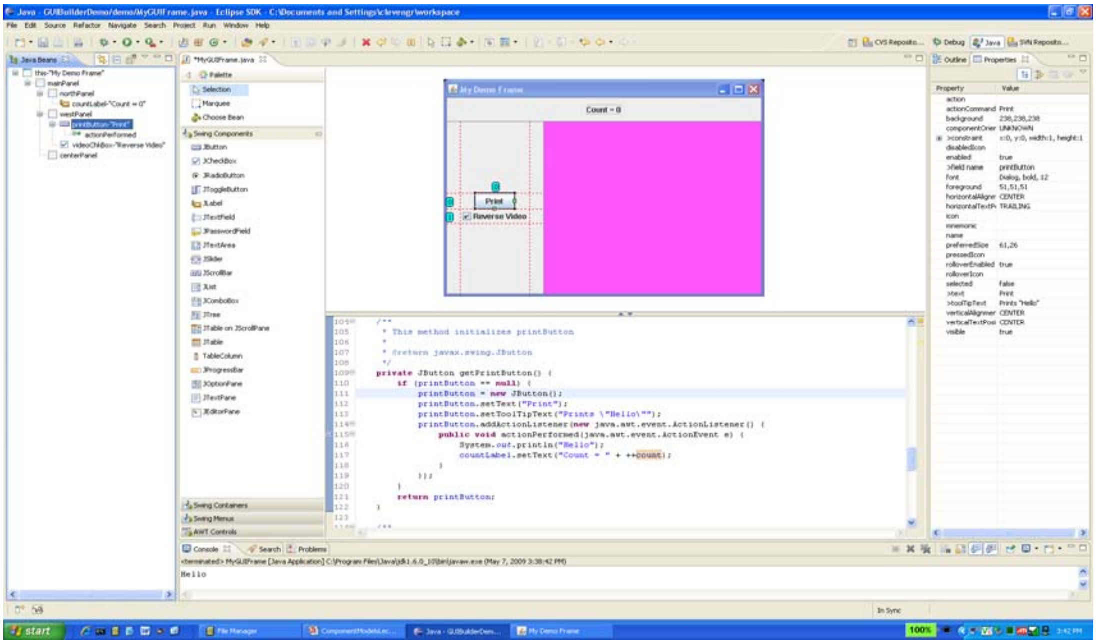
- Load a number of "plugin" classes
  - Store each loaded class, e.g. in a list/array
- Any plugin class can be *instantiated* :

```
...
Object plugin = pluginList.getNext()
Class c = plugin.getClass();
Object o = c.newInstance()
```

Additional Reflection Capabilities

- Enumerating constructors
- Invoking methods
  - Class **`Method`** contains method **`invoke()`**
- Controlling accessibility
  - Classes **`Field`** and **`Method`** contain method **`setAccessible()`**
  - Can be restricted using a **`SecurityManager`**

GUIBuilder Example



```
/**
 * This method initializes printButton
 *
 * @return javax.swing.JButton
 */
private JButton getPrintButton() {
    if (printButton == null) {
        printButton = new JButton();
        printButton.setText("Print");
        printButton.setToolTipText("Prints \"Hello\"");
        printButton.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                System.out.println("Hello");
                countLabel.setText("Count = " + ++count);
            }
        });
    }
    return printButton;
}

/**
```

Property	Value
action	
actionCommand	Print
background	238,238,238
componentOrier	UNKNOWN
>constraint	x:0, y:0, width:1, height:1
disabledIcon	
enabled	true
>field name	printButton
font	Dialog, bold, 12
foreground	51,51,51
horizontalAligner	CENTER
horizontalTextP	TRAILING
icon	
mnemonic	
name	
preferredSize	61,26
pressedIcon	
rolloverEnabled	true
rolloverIcon	
selected	false
>text	Print
>toolTipText	Prints "Hello"
verticalAlignmer	CENTER
verticalTextPosi	CENTER
visible	true

```
<terminated> MyGUIFrame [Java Application] C:\Program Files\Java\jdk1.6.0_10\bin\javaw.exe (May 7, 2009 3:38:42 PM)
Hello
```

<< This page intentionally left (almost) blank >>

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
John Clevenger / Scott Gordon

XVIII – Threads

- **Threads vs. Processes**
- **Java Threads**
- **Thread Synchronization**

CSc 133 Lecture Notes

XVIII - Threads

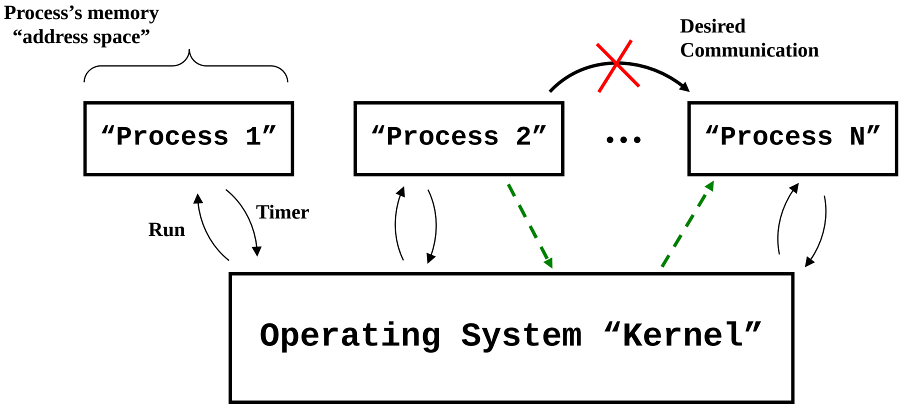
Computer Science Dept.
CSUS

Overview

- The Thread Concept
- Java Threads
  - Thread Class
  - Runnable Interface
- Thread States
- Synchronization
- Application Uses

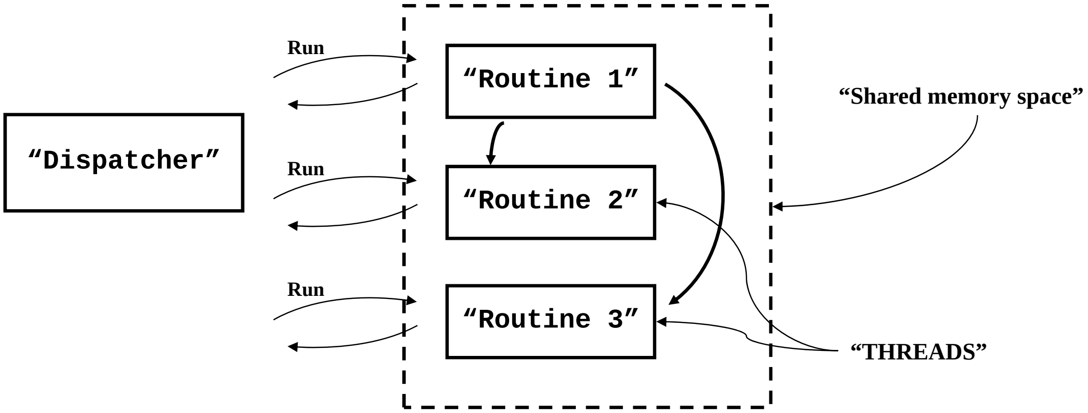
Threads vs. Processes

- OS shares CPU between “processes”
  - Processes cannot access outside their own “address space”
  - Processes can only communicate via kernel-controlled mechanisms



Threads vs. Processes

- Sometimes we *want* to allow separate pieces of code to communicate
  - Put them in the same memory space
  - Provide a mechanism to run each one independently
  - Allow the *programmer* to worry about “conflicts”



Java Threads

- Two creation mechanisms:
  - Extend class **`Thread`**
  - Construct a thread from an object that implements Interface **`Runnable`**

```
public interface Runnable
{
  void run();
}
```

```
public class MyClass implements Runnable {
  public void run() {
    ...
  }
  ...
}
...
Runnable r1 = new MyClass();
Thread t1 = new Thread(r1);
t1.start();
```

Example 1: Counter Thread

```
/** This class constructs a separate thread running a
 *  “Counter” object, starts that thread running, and
 *  waits until the thread completes before terminating.
 */
public class CounterTest {
  public static void main(String[] args) {
    // Create a "Runnable" object
    Runnable r1 = new Counter(25);
    // Construct an instance of Thread, passing the
    // Runnable as the code to be run by the Thread.
    Thread t1 = new Thread(r1);
    // Start the thread running
    t1.start();
  }
}
```

Example 1: Counter Thread (cont.)

```
public class Counter implements Runnable {
  private int loopLimit;

  public Counter(int loopLimit) {
    this.loopLimit = loopLimit;
  }

  // Specify the runnable (thread) behavior.
  public void run() {
    for (int i=1; i<=loopLimit; i++) {
      System.out.println(i);      // display current loop count
      pause(Math.random());       // sleep for up to 1 second
    }
  }

  private void pause(double seconds) {
    try {
      Thread.sleep(Math.round(1000.0*seconds));
    }
    catch(InterruptedException ie) {
      System.err.println (“Sleep interrupted”);
    }
  }
}
```

Run

Example 2: Concurrent Output

```
public class ConcurrentOutput {

  public static void main(String[] args) {

    Runnable r1 = new Counter(25);
    Thread t1 = new Thread(r1);
    t1.start();

    // cause some output from main program
    for (int i=0; i<20; i++) {
      try {
        Thread.sleep(500);
      }
      catch (Exception e) {
        System.err.println (“Sleep interrupted”);
      }
      System.out.println ("*****");
    }
    System.out.println ("Main: done.");
  }
```

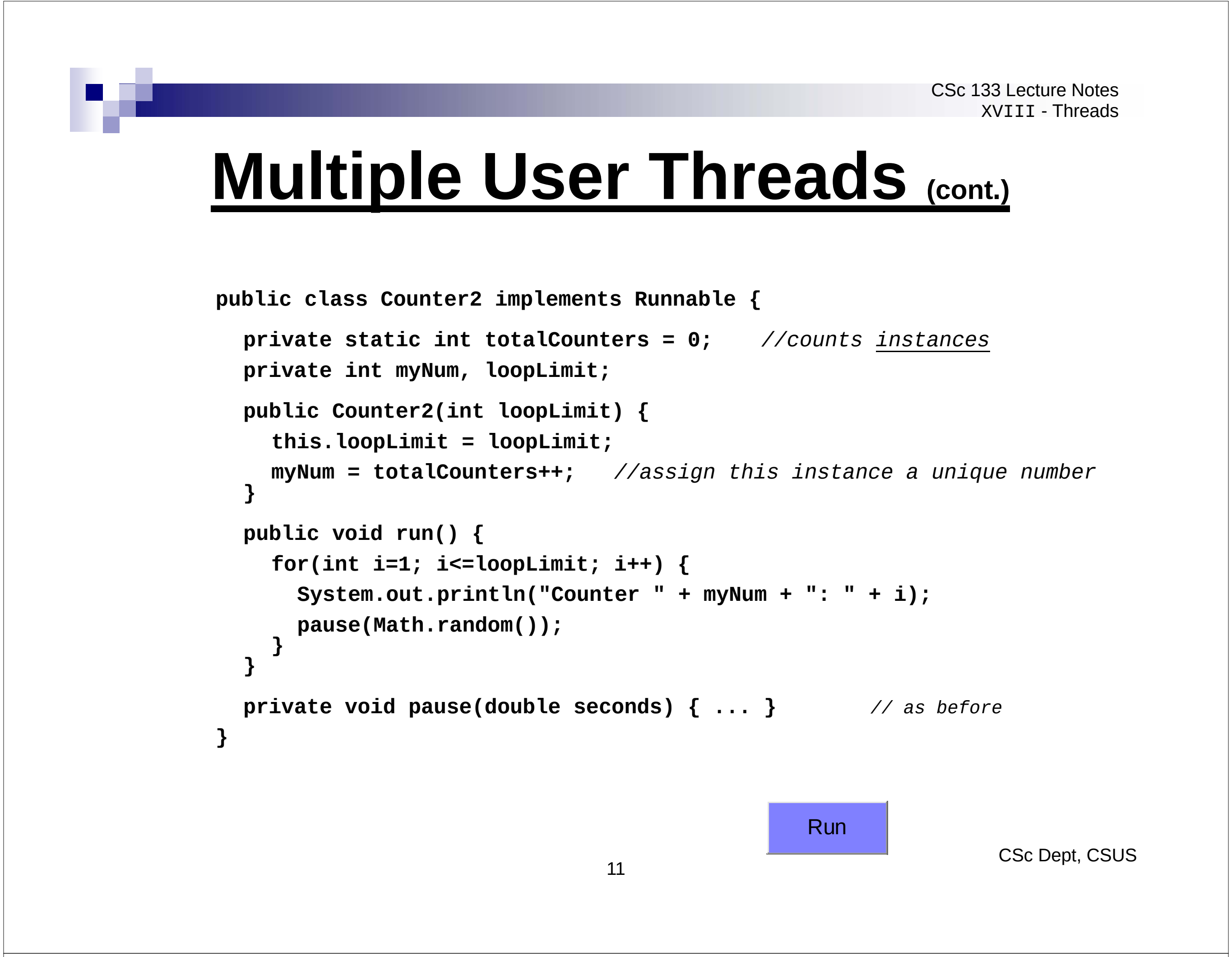
Example 2: Concurrent Output (cont.)

```
public class Counter implements Runnable {
  ... (initialization here -- same as before)
  public void run() {
    for (int i=1; i<=loopLimit; i++) {
      System.out.println(i);
      pause(Math.random());
    }
    System.out.println ("Counter: done.");
  }
  private void pause(double seconds) {
    ... as before
  }
}
```

Run

Example 3: Multiple User Threads

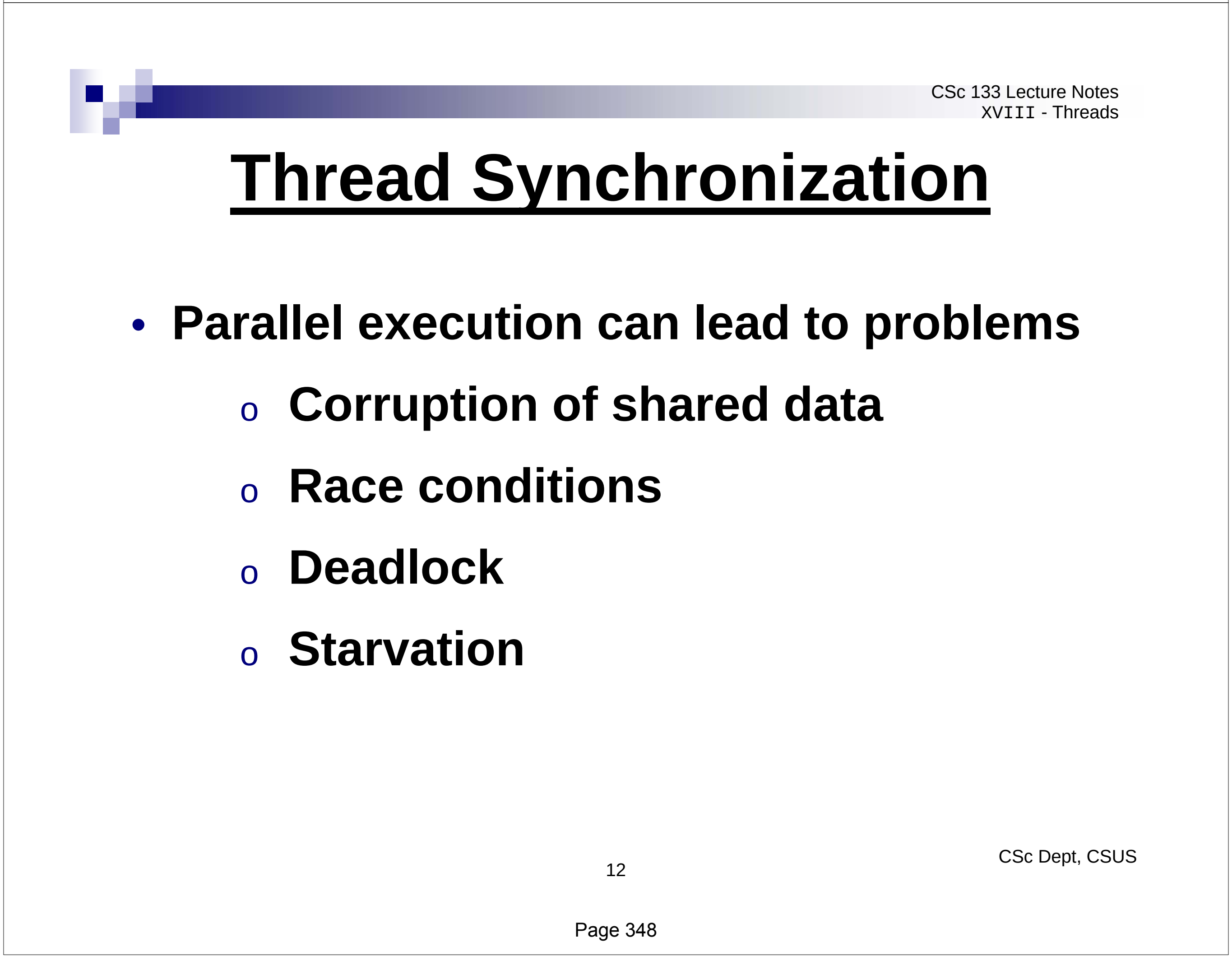
```
public class MultipleCounters {
  public static void main(String[] args) {
    /* Create multiple "Runnable" objects */
    Runnable r1 = new Counter2(8);
    Runnable r2 = new Counter2(8);
    Runnable r3 = new Counter2(8);
    /* Create threads for each runnable */
    Thread t1 = new Thread(r1);
    Thread t2 = new Thread(r2);
    Thread t3 = new Thread(r3);
    /* Start the threads running  */
    t1.start();
    t2.start();
    t3.start();
  }
}
```



Multiple User Threads (cont.)

```
public class Counter2 implements Runnable {
  private static int totalCounters = 0;    //counts instances
  private int myNum, loopLimit;
  public Counter2(int loopLimit) {
    this.loopLimit = loopLimit;
    myNum = totalCounters++;   //assign this instance a unique number
  }
  public void run() {
    for(int i=1; i<=loopLimit; i++) {
      System.out.println("Counter " + myNum + ": " + i);
      pause(Math.random());
    }
  }
  private void pause(double seconds) { ... }       // as before
}
```

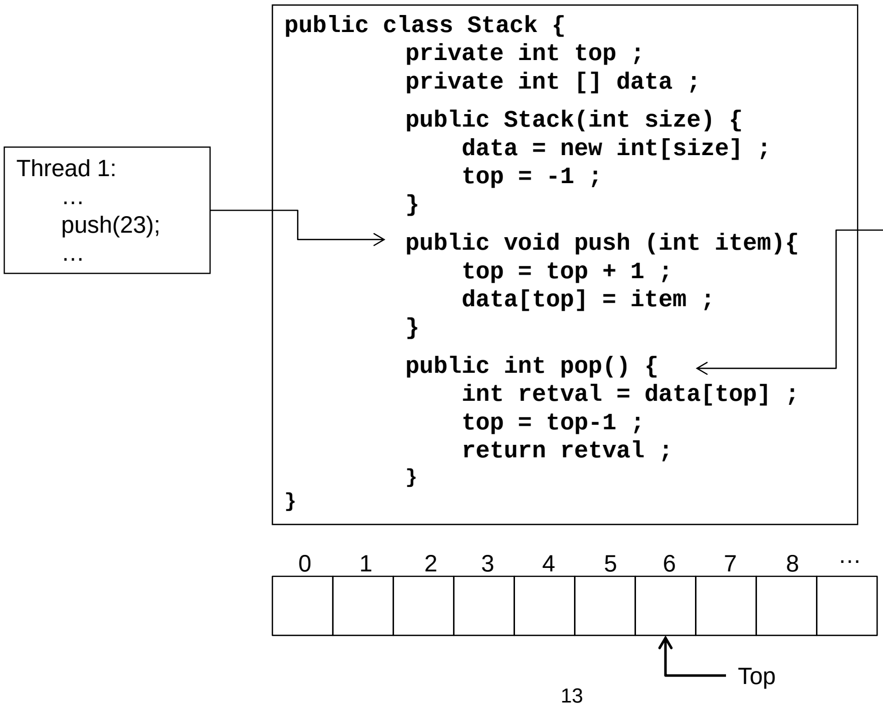
Run



Thread Synchronization

- Parallel execution can lead to problems
  - Corruption of shared data
  - Race conditions
  - Deadlock
  - Starvation

Example: Data Corruption



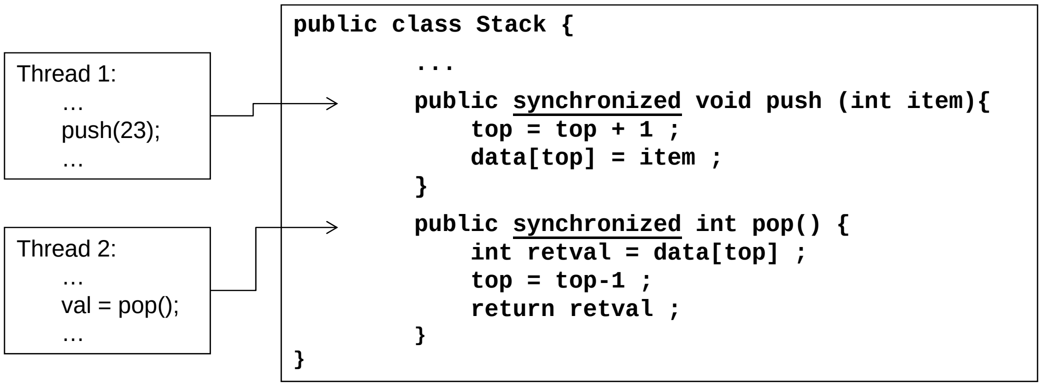
```
public class Stack {
        private int top ;
        private int [] data ;
        public Stack(int size) {
            data = new int[size] ;
            top = -1 ;
        }
        public void push (int item){
            top = top + 1 ;
            data[top] = item ;
        }
        public int pop() {
            int retval = data[top] ;
            top = top-1 ;
            return retval ;
        }
}
```

```
Thread 1:
   …
   push(23);
   …
```

```
Thread 2:
   …
   val = pop();
   …
```

Java Thread Synchronization

Example: Synchronized Methods



```
public class Stack {
        ...
        public synchronized void push (int item){
            top = top + 1 ;
            data[top] = item ;
        }
        public synchronized int pop() {
            int retval = data[top] ;
            top = top-1 ;
            return retval ;
        }
}
```

```
Thread 1:
   …
   push(23);
   …
```

```
Thread 2:
   …
   val = pop();
   …
```

```
//Another example: synchronized code blocks
public class SomeClass {
   Boolean flag = new Boolean();  // an arbitrary object
   public void someMethod() {
      ...
      synchronized (flag) { // a block of code synchronized on the "flag" object
        ... // arbitrary code here
      }
   }
}
```

Other Important Thread Methods

- **`sleep()`**
  - forces current thread to stop for a specified amount of time
- **`yield()`**
  - forces current thread to give up control to threads of equal priority
- **`currentThread()`**
  - returns the currently executing thread
- **`join()`**
  - E.g. **`myOtherThread.join()`** : waits for the other thread to die (forces a “sync point” where the threads ‘join together’)

Common Thread Uses

- Update vs. Display of Game Worlds
- Event Handling (e.g. AWT Thread)
- Image Loading
- Audio File Playing
- Pipelined Communications

Appendices

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

Elements of Matrix Algebra

by
John Clevenger

1. Definition and Representation

A *matrix* is a rectangular array of elements, arranged in rows and columns. We frequently number the rows and columns starting from zero, as shown:

$$\begin{array}{cc} & \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} X & X & X \\ X & X & X \\ X & X & X \\ X & X & X \end{pmatrix} \end{array}$$

We characterize a matrix by giving the number of rows, then columns: the example above is a *4x3* matrix. Note that by convention the number of *rows* is always given first.

In general the elements of a matrix can contain any object. However, when the elements are *numbers*, certain useful operations can be defined. The following sections describe some common matrix operations and assume the elements of the matrices in question are numbers.

2. Scalar Multiplication

One well-defined operation on a matrix is *scalar multiplication*, meaning multiplying a specific scalar value into a matrix. The result of scalar multiplication is to produce a new matrix with the same number of rows and columns as the original matrix, and where each element of the new matrix contains the product of the scalar value with the corresponding element value from the original matrix.

For example, the following shows the result of multiplying the scalar value 2 by the matrix M1, producing a new matrix M2:

$$2 \quad * \quad \overset{\text{M1}}{\begin{pmatrix} 1 & 5 \\ -2 & 4 \\ 3 & 1 \end{pmatrix}} \quad = \quad \overset{\text{M2}}{\begin{pmatrix} 2 & 10 \\ -4 & 8 \\ 6 & 2 \end{pmatrix}}$$

3. Matrix Addition

Two matrices can be added together to produce a new (third) matrix. However, this operation is only defined if both the number of rows in the first matrix is the same as the number of rows in the second matrix and also the number of columns in the first matrix is the same as the number of columns in the second matrix.

For two matrices A and B which have the same number of rows and also the same number of columns, the sum A + B is a new matrix C where C has the same number of rows and columns as A and B, and where each element of C is the sum of the corresponding elements of A and B.

For example, the following shows the result of adding two matrices A and B:

$$\overset{A}{\begin{pmatrix} 1 & 5 \\ -2 & 4 \\ 3 & 1 \end{pmatrix}} + \overset{B}{\begin{pmatrix} 2 & 2 \\ -4 & 6 \\ -3 & 1 \end{pmatrix}} = \overset{C}{\begin{pmatrix} 3 & 7 \\ -6 & 10 \\ 0 & 2 \end{pmatrix}}$$

Note that because of the definition of the elements of C (being the sum of the corresponding elements of A and B), it is the case that matrix addition is *commutative*; that is, A + B = B + A.

4. Vector Multiplication

A matrix which has only one row is sometimes called a *vector*. (This is because of the similarity to the vector algebra representation for vectors – as a single-row arrangement of vector components.) For example, the following are two different vectors (single-row matrices):

$$\begin{pmatrix} 2 & 3 \end{pmatrix} \qquad \begin{pmatrix} 4 & 6 & -1 \end{pmatrix}$$

Given a vector (single-row matrix) and another matrix, the two can be multiplied together. However, this operation is only defined if the number of elements in the vector (the number of vector "columns") is equal to the number of *rows* in the matrix by which it is multiplied.

In that case, the result of the multiplication is a new *vector* (single-row matrix) with the same number of elements (columns) as the number of <u>*columns*</u> in the <u>*matrix*</u>, and where the value of each element of the new vector is the sum of the products of corresponding elements in the original vector and the corresponding column of the matrix. This operation is known as taking the *inner product* (also called the "***dot product***") of the vector with each matrix column. Note that it is the inner product of the vector with the first matrix column that forms the first element in the result vector, and so forth.

For example, the following shows the result of multiplying a 1x2 vector V1 with a 2x3 matrix M1, producing a new vector V2. Note that V1 has 2 columns and M1 has two rows, so they met the requirements for being able to perform the multiplication operation. Note also that the new vector (V2) has the same number of elements (3) and the number of *columns* in the <u>*matrix*</u>.

$$\begin{array}{ccccc} V1 & & M1 & & V2 \\ \begin{bmatrix} 2 & 3 \end{bmatrix} & * & \begin{bmatrix} 1 & 2 & -1 \\ 4 & 5 & 0 \end{bmatrix} & = & \begin{bmatrix} 14 & 19 & -2 \end{bmatrix} \end{array}$$

It is important to note that the result of multiplying a vector by a matrix is always a new *vector*, and that the new vector always has the same number of elements (columns) as the original *matrix*. The elements of V2 above were formed by computing each of the following inner products ("•" represents the inner product or "*dot product*" operation) :

[2 3] • [1 4] = (2*1) + (3*4) = 14 ;

[2 3] • [2 5] = (2*2) + (3*5) = 19 ; and

[2 3] • [-1 0] = (2*-1) + (3*0) = -2.

5. Row-major vs. Column-major Notation

The previous section describes a vector as a matrix with a single *row*; however, a vector can also be viewed as a matrix with a single *column*. In this case the vector is written vertically, with the elements forming a column – for example:

$$\begin{bmatrix} 2 \\ 3 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 4 \\ 6 \\ -1 \end{bmatrix}$$

The difference in the two representations is simply one of convenience; the column form (also called "column-major form") of a vector does not somehow represent a "different" vector than the equivalent row-major form.

However, along with the difference in notation comes a difference in representation of the multiplication operation. The multiplication of a (row) vector by a matrix always proceeds "from the left" – that is, the vector is always written on the left side of the multiplication sign (as shown in the preceding section), and the inner products are formed between the vector and consecutive *columns* of the matrix. When a vector is represented in "column-major" form, multiplication is always written with the vector on the *right* side, and the multiplication always proceeds from *right to left*.

For column-major representation, the elements of the result vector are formed by computing the inner products of the vector with each *row* of the matrix. This therefore means that for column-major representation (multiplication from the right), the number of elements (rows) of the vector must equal the number of *columns* in the matrix (as opposed to being equal to the number of rows in the matrix, which is the rule for row-major representation and multiplication from the left).

The form for column-major representation and multiplication from the right is shown below. Note that in this example the initial vector is written on the *right*, and the number of *columns* in the matrix matches the number of elements (rows) in the initial vector. Note also that the result (which is a vector, as before) is formed by computing the inner product of the initial vector with each *row* of the matrix (instead of with each column of the matrix as is done with row-major representation and multiplication from the left).

$$\overset{V2}{\begin{pmatrix} 14 \\ 19 \\ -2 \end{pmatrix}} = \overset{M1}{\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ -1 & 0 \end{pmatrix}} * \overset{V1}{\begin{pmatrix} 2 \\ 3 \end{pmatrix}}$$

The form which is used for vector multiplication (row/from the left or column/from the right) is mostly a matter of preference and notational convenience. Mathematics textbooks tend to use column-major form, while computer graphics texts tend to be split evenly between row-major and column-major form. Programming languages which support matrix operations tend to use column-major form. Regardless of which notation is used, it is important to be consistent, and it is also of critical importance to compute the inner products based on the representation given.

6. Transpose

The *transpose* of a matrix A is formed by “exchanging” the rows and columns of A such that for every element (i,j) in the original matrix, the same value is found at element (j,i) in the new matrix. That is, the value at row 2, column 1 is exchanged with the value at row 1, column 2, and so forth. This generates a new matrix called the *transpose* of A, denoted $A^{\mathbf{T}}$. The following shows an example of a matrix A and its transpose $A^{\mathbf{T}}$.

$$A = \begin{pmatrix} 2 & 10 & -3 \\ -4 & 8 & 4 \\ 6 & 2 & -7 \end{pmatrix} \qquad A^{\mathbf{T}} = \begin{pmatrix} 2 & -4 & 6 \\ 10 & 8 & 2 \\ -3 & 4 & -7 \end{pmatrix}$$

Note that for a square matrix (such as A, above), transposing the matrix has the effect of “flipping” it along the diagonal running from upper left to lower right (called the “major diagonal’). However, a matrix need not be square to form the transpose, as shown in the next example:

$$B = \begin{pmatrix} 2 & 10 \\ -4 & 8 \\ 6 & 2 \end{pmatrix} \qquad B^{\mathbf{T}} = \begin{pmatrix} 2 & -4 & 6 \\ 10 & 8 & 2 \end{pmatrix}$$

Note that transposing a non-square matrix has the effect of producing a new matrix whose number of *rows* is equal to the number of *columns* in the original matrix (and vice versa).

Another important point to note is that in the discussion of row-major vs. column-major representation (above), switching from multiplication on the left (with the vector on the left) to multiplication on the right (vector on the right) essentially involves forming the transpose of the matrix which is being multiplied (compare the examples in the two preceding sections to confirm this). That is, multiplying a row vector by a matrix "from the left" produces an equivalent result to multiplying the column form of the same vector "from the right" into the transpose of the matrix.

Two important identities relating matrix transposes are:

1. $(A + B)^{\mathbf{T}} = A^{\mathbf{T}} + B^{\mathbf{T}}$; that is, the transpose of a sum of two matrices equals the sum of the transposes of the individual matrices.

2. $(A * B)^{\mathbf{T}} = B^{\mathbf{T}} * A^{\mathbf{T}}$; that is, the transpose of a matrix product A*B is the product of the transpose of B times the transpose of A. Note that since matrix multiplication is *not* commutative (see below), the order of this result is important.

7. Matrix Multiplication

The generalized form of matrix multiplication is to multiply a matrix A by a second matrix B (with A on the left and B on the right) producing a third matrix C. However, this operation is only defined if the number of *columns* of the first matrix A is the same as the number of *rows* of the second matrix B. In such a case, matrices A and B are said to be *conforming* matrices.

The result of multiplying two conforming matrices A * B is a new matrix C which has the same number of *rows* as A and the same number of *columns* as B. That is, if A is an *m x p* matrix (i.e, has "m" rows and "p" columns), and B is a *p x n* matrix ("p" rows and "n" columns), then the product A*B produces a new matrix C which is *m x n.* Further, each element (i,j) of C is the scalar value produced by the inner product of the i^{th} row of A with the j^{th} column of B.

For example, multiplying the following *2 x 3* matrix A by the *3 x 4* matrix B produces the *2 x 4* matrix C as shown:

$$\begin{array}{ccccc} A & * & B & = & C \\ \begin{pmatrix} 2 & 3 & -1 \\ 4 & 0 & 6 \end{pmatrix} & * & \begin{pmatrix} 1 & -1 & 0 & 4 \\ 2 & -2 & 1 & 3 \\ 5 & 7 & 1 & -3 \end{pmatrix} & = & \begin{pmatrix} 3 & -15 & 2 & 20 \\ 34 & 38 & 6 & -2 \end{pmatrix} \end{array}$$

Each element of C in the above example is formed by computing the inner product of a row of A with a column of B. For example, the element with value “3” in row 0, column 0 of C is formed by the inner product of row 0 of A with column 0 of B: (2*1) + (3*2) + (-1*5) = 3. Likewise, the element with value “6” in row 1 (the second row), column 2 (the third column) of C is formed by the inner product of row 1 of A with column 2 of B: (4*0) + (0*1) + (6*1) = 6. Each of the other elements of C is likewise formed by computing the inner product of a row of A with a column of B.

Note that because the matrix multiplication operation is only defined when the number of columns of the first (left) matrix equals the number of rows of the second (right) matrix, in general it is *not* true that just because A*B is a defined operation then it follows that B*A is also well defined. In the example above, for instance, A*B is defined (and produces the matrix C as shown), but the operation B*A *is not defined* – because B does not have the same number of columns as the number of rows in A.

The only time that both A*B and B*A are well-defined matrix multiplication operations is when both A and B are *square* matrices of the *same size*.

Note that just because A and B are square matrices of the same size (and hence both A*B and B*A are well-defined), it is not in general true that A*B = B*A. That is, *the matrix multiplication operation is not commutative*. This property (lack of commutativity of matrix multiplication) is important to keep in mind when manipulating matrices.

8. Identity Matrix

We define a special matrix form called the *Identity matrix*. The Identity matrix has the properties that (1) it is square; (2) it has 1’s in every element along its major diagonal; and (3) it has zeroes in every element not on its major diagonal. For example, the following shows an Identity matrix I of size 3:

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Identity matrices have certain important and useful properties. For example, if I is an identity matrix of size *n*, and A is also a square matrix of size *n*, then it is the case that

$$A * I = I * A = A.$$

That is, multiplying matrix A by the identity matrix, whether on the left or on the right, leaves A unchanged (this is in fact the mathematical definition of an “identity element”; for example, the value “1” is the identity element with respect to the ordinary algebra operation “multiplication” – multiplying a given number by 1 does not change its value).

9. Matrix Inverses

A given matrix "M" is said to be *invertible* (or "*has an inverse*"), if there exists another matrix, denoted as M^{-1}, such that

$$M * M^{-1} = M^{-1} * M = I,$$

where I is the Identity matrix of the same size as M. The following general properties relate to matrix inverses:

(1) only *square* matrices have inverses (this is a consequence of the above definition, which requires the same result for multiplication from the left and right);

(2) not all square matrices have inverses (that is, there are some matrices for which, even though they are square, it is not possible to find another matrix for which the above equations hold); and

(3) if a matrix has an inverse, it will be *unique* (that is, every matrix has at most one inverse).

An important observation about matrix inverses is that they represent, in a sense, an "opposite" operation. That is, if M represents some operation (say, a transformation of some kind), then M^{-1} represents the "opposite transformation". This observation is useful when attempting to build a series of transformations which are represented by matrices; it provides a method of specifying how to "undo" a given operation. However, care must be taken to insure that the operation in question (being represented by a matrix) is "undoable" (i.e. that the matrix is invertible). See the section on "Inverse Of Products" for further information on this topic.

10. Associativity and Commutativity

As noted above, the matrix multiplication operation is *not* commutative. That is, it is *not* true (in the general case) that A*B produces the same result (matrix) as B*A.

However, an important property of matrix multiplication is that it <u>is</u> *associative*. That is, given three matrices A, B, and C, multiplied from left to right, the same result will be obtained whether the left or right multiplication is performed first. That is,

$$(A * B) * C = A * (B * C)$$

This associative property of matrix multiplication allows matrix operations to be combined in various different ways for efficiency.

11. Inverse of Products

A useful theorem for manipulating matrices is that *the inverse of a matrix product is equal to the product of the inverses of the individual matrices, multiplied in the opposite order.* For example, suppose we have the matrix product (A * B * C), which as noted above can be computed as ((A * B) * C) or (A * (B*C)) due to associativity of matrix multiplication. The inverse of this product is denoted $(A * B * C)^{-1}$. Then the above theorem says that

$$(A * B * C)^{-1} = C^{-1} * B^{-1} * A^{-1}.$$

That is, the inverse of (A*B*C) can be obtained by first obtaining each individual inverse matrix (A^{-1}, B^{-1}, and C^{-1}), and then multiplying those matrices together in the opposite order.

Note that the product on the right-hand side is written in the opposite order from the one on the left, and that *this order is significant since matrix multiplication is* ***not*** *commutative* (even though it is associative).

The inverse-of-products theorem is useful in situations where you have a series of transformations represented in matrix form and you want to "undo" the transformations – that is, you want to find a transformation which goes in the "opposite direction". If the original series of transformations is A, then B, then C, then the "opposite" transformation would be that which "undoes (A, then B, then C)" – that is the *inverse* of (A * B * C). By the inverse-of-products theorem, this "opposite" transformation is exactly the transformation ($C^{-1} * B^{-1} * A^{-1}$). (Note: in order for this to work, it must be true that each individual matrix (A, B, and C) is itself invertible.

12. Order of Application of Matrix Transforms in Code

Matrices can be used to represent "transformations" to be applied to objects. For example, a single matrix can represent a "translation" operation to be applied to a "point" object. Applying matrix transformations correctly in program code requires a thorough understanding of the rules of matrix algebra regarding associativity, commutativity, and inverse products (discussed above), as well as an understanding of how code statements translate into matrix operations.

Suppose for example that you wish to apply a translation, represented by a matrix, to a given point P1. This is done by multiplying the point (represented as a vector as described above) by the matrix, which results in a new (translated) point P2. As noted above, matrix multiplication can be done either using "row-major" form or "column-major" form. However, most programming language matrix libraries (including those in Java) use column-major form – that is, matrix multiplication is done "from the right". Assuming we are using this convention (for example, assuming we are writing in Java), then when you multiply a point by a transformation matrix, you do so "from the right", meaning that the point is written on the right hand side with the matrix to its left, and the multiplication proceeds that way ("from right to left").

The algebraic representation of the above example would be the following, where "P1" is the original point and "M" is the matrix containing the desired translation:

$$\begin{array}{ccccc} P2 & & M & & P1 \\ \begin{bmatrix} X' \\ Y' \end{bmatrix} & = & \begin{bmatrix} \quad & \quad \\ \quad & \quad \end{bmatrix} & * & \begin{bmatrix} X \\ Y \end{bmatrix} \end{array}$$

Suppose for example that the original point P1 was (1,1) and the matrix contained a translation of (1,1). Then the Java code to accomplish this would be:

```
Point p1 = new Point(1,1);
AffineTransform at = new AffineTransform();
at.translate(1, 1);

Point p2;
at.transform(p1,p2);
```

"**`AffineTransform`**" is the Java class representing transformation matrices; in fact, we sometimes for convenience just refer to objects of type **`AffineTransform`** as "matrices". **`translate()`** is a method which causes the specified matrix (**`AffineTransform`** object) to contain the specified translation, and **`transform()`** is a method which multiplies its first argument by the matrix and returns the result in the second argument, with the multiplication being applied "from the right". In the above example the original point P1 has a value of (1,1) and the matrix has a translation of (1,1), so the value of P2 after executing the above code will be (2,2) since applying a translation of (1,1) to a point with value (1,1) results in a point with value (2,2).

A transformation matrix contains (in the general case) multiple transforms -- e.g. a scale, a rotate, a translate, another rotate, etc. These transformations exist (conceptually) in the matrix *in some order* -- that is, there is one which is "rightmost", one immediately to its left, one immediately to THAT one's left, etc. The order in which the transformations exist in the matrix is determined by the code which inserts them into the matrix. Inserting a transformation is done by multiplying the new transformation "on the right" of the existing transformation, so the new transformation exists "on the right" in the new compound transformation matrix.

For example, the following Java code creates a transformation matrix (**`AffineTransform`** object) containing TWO transformations: a rotation by 90° and a translation by (1,1):

```
AffineTransform at = new AffineTransform();
at.translate(1, 1);
at.rotate(Math.toRadians(90));
```

Initially (that is, when it is first created) the matrix contains the Identity transformation. Since the first method invoked on the matrix is `translate()`, the specified translation becomes the "leftmost" (and in this case the only) transformation in the matrix. Since the `rotate()` is called *after* the `translate()`, the rotation is concatenated "on the right" in the matrix; that is, the rotation is the rightmost transformation in the matrix and the translation is to the left of the rotation.

When you multiply a point by a matrix, you are applying the transformations contained in the matrix to the point "in order, from right to left". That is, the rightmost transformation gets applied first, then the one to its left, then the next leftward one, etc. For example, consider the following Java code:

```
Point p1 = new Point(1,1);
AffineTransform at = new AffineTransform();
at.translate(1, 1);
at.rotate(Math.toRadians(90));

Point p2;
at.transform(p1,p2);
```

The value in P2 after this code is executed will be (0,2). This is because the `transform()` method multiplies the point P1 by the matrix *from the right*, causing the point to first be transformed by the rightmost transformation in the matrix (the rotation) – which rotates the original point (1,1) to the point (-1,1) – then transformed by the translation, which translates the point (-1,1) by (1,1) resulting in the value (0,2) in P2. (If this is not clear you should draw the points on graph paper, construct the matrix containing the two transformations by hand, and convince yourself that the above statements are correct.)

Note that it is the case (as shown in the example above) that *the order of **application** of transformations contained in a matrix is the **opposite of the order in which the code statements defining the transformations are executed***. In the above example the `translate()` method was called *before* the `rotate()` method, meaning that when the `transform()` method was invoked to multiply the point by the matrix the *rotation* was applied *before* the translation.

Note that if the above example had instead been written as:

```
Point p1 = new Point(1,1);
AffineTransform at = new AffineTransform();
at.rotate(Math.toRadians(90));
at.translate(1, 1);

Point p2;
at.transform(p1,p2);
```

(that is, if the rotation had been concatenated into the matrix *first*), then the result would have been different – specifically, the final value in P2 would have been (-2,2), since the `transform()` method would have the effect of applying the translation first (translating the original point (1,1) to (2,2)) and then applying the rotation second (rotating the point (2,2) to (-2,2)). Note also that this difference is a manifestation of the algebraic rule that matrix multiplication is not commutative, as discussed above.

Therefore, if you have a certain order in which you want a set of tranformations *applied*, you must get them into the transformation matrix such that they are in order, from *right* to *left*, in the order that you want them applied. If you have, say, a scale and a translate to be applied, and you want the translate to be applied first, it must be on the rightmost side of the matrix, with the scale to its left. This in turn means that you must put the SCALE in the matrix *first*, and THEN put the translate in the matrix so that the translate is on the right (and the scale to its left). This means you must call the AffineTransform `scale()` method in your code BEFORE you call the AffineTransform `translate()` method. Writing code to apply matrix transformations therefore involves first figuring out the order in which you want the transforms applied, and then writing code statements (in the proper order) that cause those transforms to end up in the matrix in the right-to-left order you need.

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

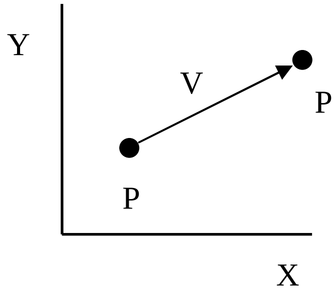
Elements of Vector Algebra

by
John Clevenger

1. Definition and Representation

A *vector* is an object with *magnitude* and *direction.* A vector is commonly represented as an arrow, with the length of the arrow representing the magnitude, and the direction of the arrow representing the vector direction. The starting point of the arrow is called the *tail* of the vector; the ending point (arrowhead) is called the *head* of the vector.

A vector runs from one point to another, and can be represented as the *difference* between the two points. The "difference between two points" is obtained by taking the difference between corresponding elements of the two points. Thus, in 2D:



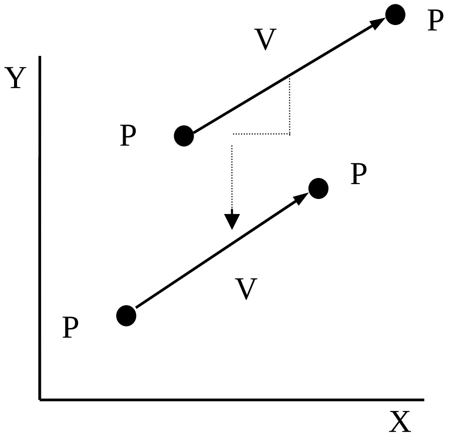
$$V = (P2 - P1)$$

$$= [\ (X_2\text{-}X_1)\ \ (Y_2\text{-}Y_1)\ ]$$

Note that a vector is represented as an object in square brackets; the elements inside the square brackets are called the *components* of the vector. Each component of a vector is a numerical quantity.

2. Translation

Vectors can be *translated* without altering their value (since they consist of *magnitude* and *direction* but not *position*):



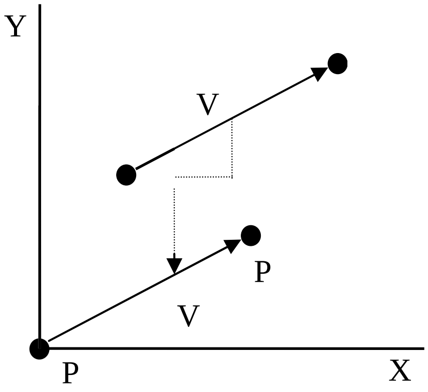
V1 = (P2 - P1)

V2 = (P4 - P3)

P4 - P3 = P2 - P1;

hence V2 = V1

Since a vector can be translated anywhere without changing its value, it follows that a vector can be translated so that its tail is at the origin:



$$V = (P2 - P1)$$

$$= [\ (X_2 - X_1)\ \ (Y_2 - Y_1)\ ]$$

$$= [\ (X_2 - 0)\ \ (Y_2 - 0)\ ]$$

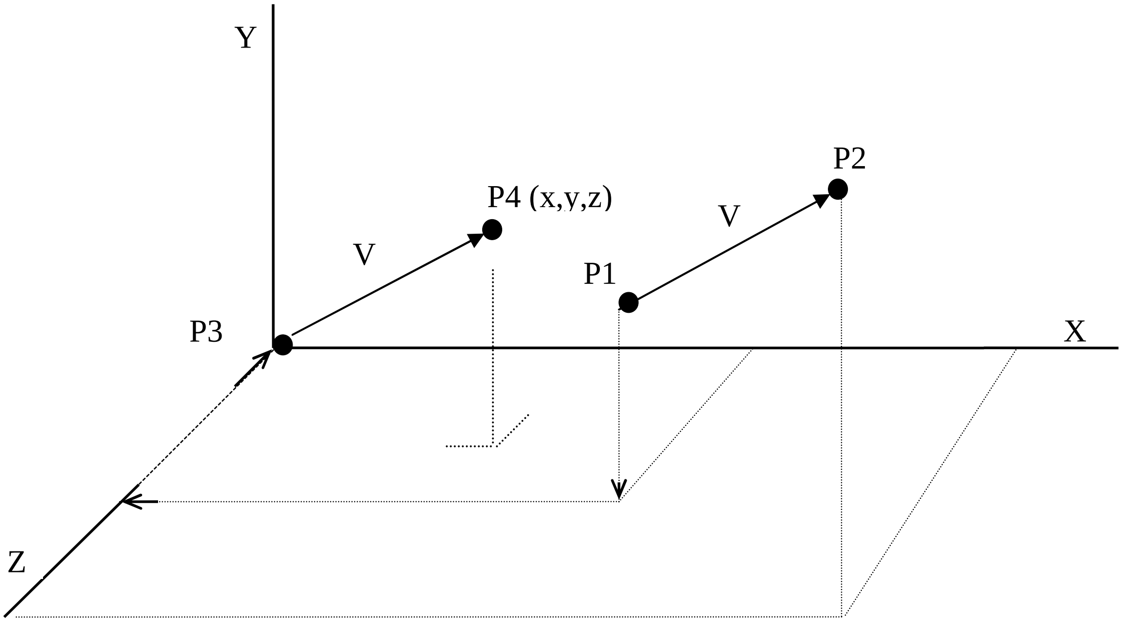
$$= [\ (X2)\ \ (Y2)\ ]$$

$$= [\ X2\ \ Y2]$$

Thus, a vector can also be represented as a *single point*: the point at the head of the vector when the tail is at the origin. Both representations of vectors (as the difference of two points or as a single point) are useful.

3. Two-Dimensional vs. Three-Dimensional Vectors

Vectors in 3D work the same way as in 2D. They can be specified as the difference between two (3D) points; they can be translated without changing their value; and they can be specified as a single (3D) point since the tail can be translated to the origin. (In fact, a 2D vector can be considered to be a special case of a 3D vector, with the Z component equal to zero.)



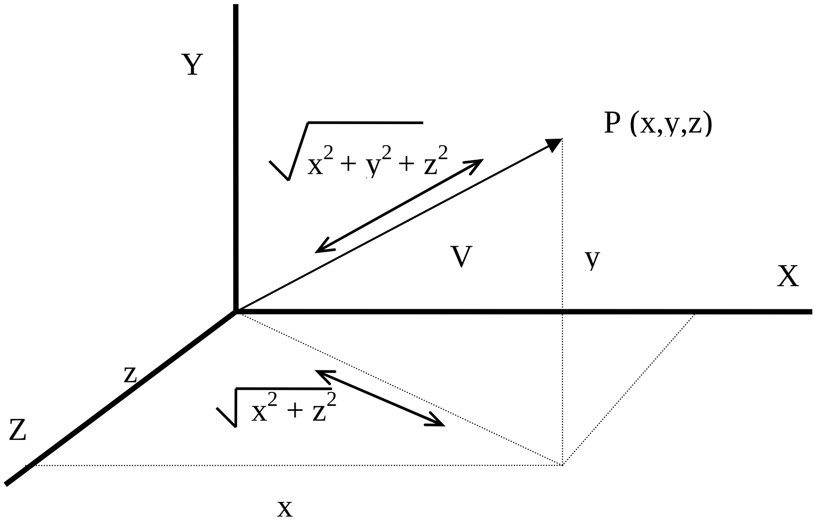
In the preceding figure,

$$\begin{aligned} V &= (P2 - P1) \\ &= (P4 - P3) \\ &= [(X_2 - X_1)\ \ (Y_2 - Y_1)\ \ (Z_2 - Z_1)] \\ &= [\ (X_4 - X_3)\ \ (Y_4 - Y_3)\ \ (Z_4 - Z_3)] \\ &= [\ (X_4 - 0)\ \ (Y_4 - 0)\ \ (Z_4 - 0)\ ] \\ &= [X_4\ \ Y_4\ \ Z_4] \\ &= [x\ \ y\ \ z] \end{aligned}$$

Note therefore that a *single point* (x, y, z) in 3 dimensions represents the 3D vector **[x y z]** , the vector from the origin to (x, y, z).

4. Magnitude

The *length* or *magnitude* of a vector V is denoted by the symbol |V|, and can be calculated using the Pythagorean Theorem:

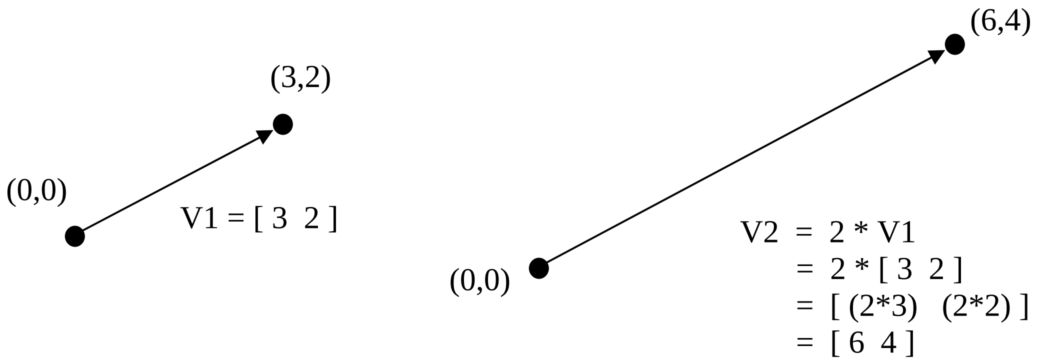


The magnitude (length) of V is $|V| = \sqrt{(x^2 + y^2 + z^2)}$.

Note again that we can consider 2D a special case of 3D where Z=0; in that case the magnitude of a 2D vector is just $|V| = \text{sqrt}(x^2 + y^2)$.

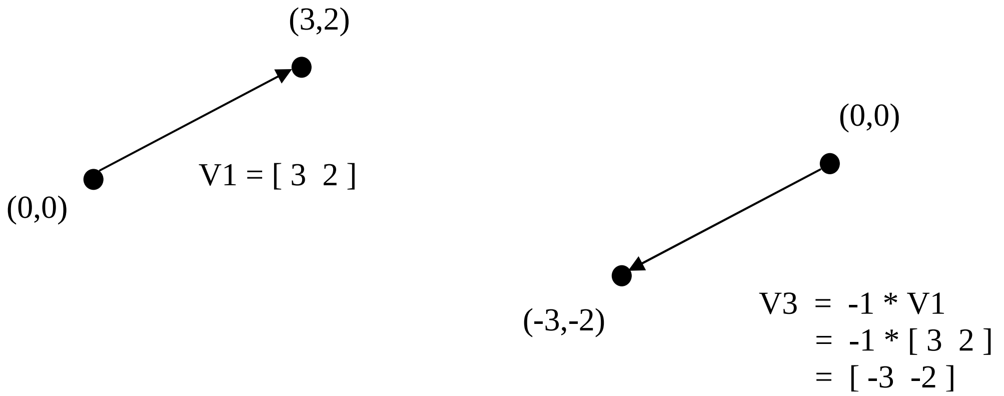
5. Scalar Multiplication

Multiplying a vector by a scalar value is a well-defined operation, and produces a new vector whose components are the result of multiplying each of the original vector components by the scalar value. For example, multiplying the 2D vector V1 = [3 2] by the scalar value 2 produces the new vector V2 = [6 4]. This is shown graphically in 2D as:



Note that the effect of multiplying a vector by a positive scalar value is to produce a new vector whose direction is the same as the original vector and whose magnitude varies according to the scalar value: multiplying by a value greater than one increases the magnitude, while multiplying by a fractional value decreases the magnitude.

Note also that multiplying a vector by a *negative* scalar value has the effect of reversing the direction of the vector. For example, using the vector V1 = [3 2] (as shown above), a new vector V3 = -1 * V1 is [-3 -2], as shown below (keep in mind that vectors can be translated without changing their value) :



6. Unit Vectors

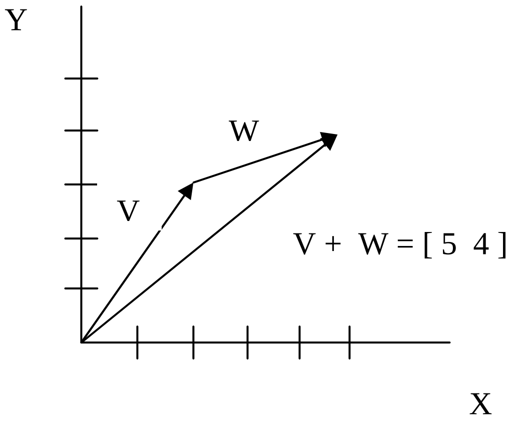
Given a vector V, there exists a corresponding vector U (also sometimes denoted as V with a “^” over it) called a *unit vector* with the property than U has the same *direction* as V but has *length = 1.*

The unit vector U for a given vector V is calculated by dividing each of the components of V by the magnitude (length) of V: U = V / |V| = [(V_x / |V|) (V_y / |V|]. For example, if V = [3 4], then |V| = sqrt($3^2 + 4^2$) = 5, and U = [(3/5) (4/5)] = [0.6 0.8]. Note that this is a vector in the same direction as V, and whose length is |U| = sqrt ($0.6^2 + 0.8^2$) = sqrt (0.36 + 0.64) = sqrt (1.0) = 1.0.

7. Vector Addition

Given two vectors V and W, the vector sum V+W is a well-defined operation and produces a new vector whose components are the sums of the corresponding components of V and W. For example, if V = [2 3] and W = [3 1], then the vector V+W = [(2+3) (3+1)] = [5 4].

Vector addition can be represented graphically by placing the tail of one vector at the head of the other; the sum will be the vector from the tail of the first vector to the head of the second. For example, using the vectors V and W given above:



8. Dot Product

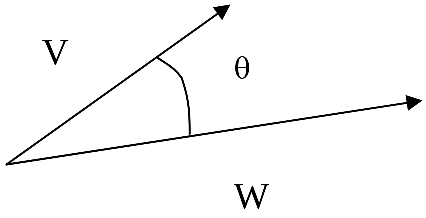
Another well-defined operation on vectors is called the *dot product* (also called the *inner product).* Given two vectors V and W, the dot product is written notationally as V • W and is defined as a scalar value D = Σ (V_i * W_i) over all components “i” of V and W. Note that the dot product operation produces a *scalar* value D – i.e., a single numeric value.

For example, if V = [2 3] and W [7 -1], then the dot product

V • W = ((2 * 7) + (3 * -1)) = (14 – 3) = 11. As noted, this result is a scalar value.

An interesting and useful property of the dot product of two vectors is that it produces the same value as the product of the magnitudes of the two vectors multiplied by the cosine of the smaller of the angles between the two vectors. That is, given two vectors V and W,

$V \bullet W = |V| * |W| * \cos(\theta)$, where θ is the angle between V and W:



Since both the magnitudes of the vectors and the scalar value of the dot product of the vectors can be easily calculated (see above), this means it is straightforward to find the angle between two vectors:

$$\cos(\theta) = (V \bullet W) \; / \; (|V| * |W|)$$

Note that if V and W are *normalized* (that is, they are Unit Vectors of length 1), then this formula reduces to

$$\cos(\theta) = (V \bullet W)$$