California State University, Sacramento
Department of Computer Science

CSC 133

Object-Oriented Computer Graphics Programming

Lecture Note Slides

Fall 2017

CSC 133

Object-Oriented Computer Graphics Programming

Contents of Lecture Note Slides

CSc 133 Lecture Notes

1 – Course Introduction

Computer Science Department
California State University, Sacramento

Overview

- Classroom conduct
- Prerequisites
- Course topics
- Texts and references
- Grading: exams and programs
- Communication
- Workload
- Ethics

Contacting Your Instructor

Dr. Pinar Muyan-Ozcelik

Office: Riverside Hall 5008

Phone: 278-6713

Office Hours: Tuesday 3:00 –3:50 pm
Thursday 1:40 – 3:50 pm
and by appointment

Email: pmuyan@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

and please refrain from:

browsing, facebooking, social networking, texting, instant messaging, tweeting, blogging, gaming, 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 (review "Java Basics" in Appendices)

- 3 semesters in Java, C++, 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 (review "Vector/Matrix Algebra" in Appendices)

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

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

What is this course about ?

Two main topics:

Fundamentals of the "O-O" paradigm

Introduction to Computer Graphics

Also covers:

Mobile App Development

First topic: **Object-Oriented Paradigm**

We will focus on how to write programs correctly!

- Language implementation:
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

- Tools supporting OOA/OOD/OOP:
  - formalisms such as *UML*
  - ***design patterns (underlying theme of CSC 133!)***

Second topic: **Computer Graphics**

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

Additional topic: **Mobile App Development**

- Introduction to Mobile App Development and CN1 (Codename One: Java-based, cross-platform mobile app development environment)
- Application of OOP and CG concepts to CN1:
  - CN1 code snippets will be provided in lectures
  - Assignments are required to be solved using CN1

Texts and References

- Required Texts:
  - CSc 133 Lecture Notes, Fall 2017, available at the "Content" section of SacCT
  - Codename One Developer Guide:

CN1 Developer Guide - Revision 3.6 (pdf is available at SacCT)

  - Codename One JavaDocs of APIs: https://www.codenameone.com/javadoc/index.html

Texts and References (cont.)

- 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
- Supplemental material:
  - Basic Debugging With Eclipse:
    https://www.youtube.com/watch?v=PJWtO5wrptg

Grading

- Weighted Curve based on:
  - Programming Assignments (4) 40%
  - Midterm Exam 25%
  - Final Exam 30%
  - Attendance 5%
- Additional Criteria
  - Passing completion of **both**:
    - Programming assignments (combined)
    - Exams (Midterm + Final combined)

Grading (cont.)

Programming Assignments

- Required to be solved using CN1, submitted via SacCT
- Important tips will be given in class!
- There will be four (4) programming assignments
- They will be ***cumulative!*** Don't try to skip one!
- Late assignments are accepted **up until 10 days** past due date
- Late penalty: 5% per day, weekend days and holidays are counted
- Submissions can be updated **only** prior to the due date**:**
  - The version submitted right before the due date will be graded
  - If no such version exists, the version submitted right after the due date will be graded (as late assignment)
- *Individual* work
- Must keep a *backup* (machine-readable) copy

Grading (cont.)

Exams

- Dates are noted on the outline
- 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*

Computers

- Work on any school machine or your machine which have CN1
- To install CN1:
  - Install latest version of Java SE JDK
  - Install latest version of Eclipse for Java Developers
  - Install CN1 as a plugin to the Eclipse

  (installation instructions will be discussed in class)

Communication

- SacCT:
  - assignments
  - announcements (discussion board)
  - feedback and grades
- Check your SacLink 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 SacCT

Ethics (cont.)

- You are allowed and encouraged to discuss assignments with other students in the class. Getting verbal advice/help from people who've already taken the course is also fine.
- Any reference to assignments from previous terms or web postings is unacceptable
- Any copying of non-trivial code is unacceptable
  - Non-trivial = more than a line or so
  - Includes reading someone else's code and then going off to write your own.

Questions?

CSc 133 Lecture Notes

2 – Introduction to Mobile App Development and CN1

Computer Science Department
California State University, Sacramento

Overview

- Why to Use a Mobile Programming Environment?
- Why to choose Codename One (CN1)?
- CN1 Features
- CN1 vs Java
- CN1 Installation
- CN1 Hello World App
- CN1 and Assignments
- Assignment#0
- CN1 Online resources

Why to Use a Mobile Programming Environment?

- Mobile computing is **ubiquitous** and allows:
  - Instant retrieval of information
  - Constant communication
  - Easy access to games, company products etc.
- Hence, there is an **ever growing need** for mobile app developers.
- Also, knowing how to program in this contemporary environment is **fun** and **cool**!

Why to Use a Mobile Programming Environment? (cont.)

- CSC 133 topics are widely applicable to a mobile programming environment.
- Hence, using this environment in the **lectures** and **assignments**, will help to:
  - Enhance learning by relating CSC 133 topics to their **contemporary use cases**
  - Provide a base for **further exploration** of this environment (apply it to other CSC topics or create your own brilliant app!)
  - Build a stronger **resume**

Why to choose Codename One (CN1)?

- There are various popular mobile programming environments:
  - Platform specific:
    - e.g. Android, iOS SDK
  - Cross-platform (write one program and run it on various platforms - iOS, Android, Windows, etc.):
    - e.g. Codename One (CN1), PhoneGap, Appcelerator, Xamarin
- We choose CN1, because it is:
  - Java-based
  - Cross-platform

CN1 Features

- Features we will use:
  - Free and open source
  - Have **simulator** environment

  (does not require you to have a mobile device)
- Features that we will **not** use:
  - Build and cloud **servers** (converts the CN1 code to a native app e.g. Android, iOS, Windows app)
  - **GUI builder** (provides drag and drop tools to automatically create GUI components)

CN1 vs Java

- CN1 API was initially limited to subset of Java 1.3 and then added support for subset of Java 5 and now supports some Java 8 language features.
- Does not support Java features that are not suitable for mobile devices e.g.:
  - Reflections
  - Desktop APIs such as java.net, java.io.File etc. (provides its own alternatives)
  - Swing library (provides Swing redesigned for mobile environment in its UI API/package)

CN1 Installation

- Install latest version of Java SE JDK (version 8, release 1.8.0_144):

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

- CN1 can be installed to one of the following IDEs: Eclipse, NetBeans, or IntelliJ IDEA which run on various operating systems.
- For 133, “Eclipse IDE for Java Developers - Oxygen package (version R)” is **required** :

http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/oxygen

- Windows is recommended.

CN1 Installation (cont.)

- Install CN1 plugin (latest version 3.7) to Eclipse using instructions at:

http://www.codenameone.com/download.html

In Eclipse select “Help” → “Eclipse Marketplace” → search for “Codename One” and follow the installation process

- Alternative Eclipse installation steps:
  - Select “Help” → “Install New Software”.
  - Click the ”Add” button on the right side.
  - Name = ”Codename One” and location = https://codenameone.com/files/eclipse/site.xml .
  - Select the entries & follow the wizard to install

CN1 Installation (cont.)

- Eclipse and CN1 are installed at:
  - ECS Open Labs [RVR 2011, SCL 1234, SCL 1208 (24 hour lab)]
  - ECS Teaching Labs [ARC 1014/1015 (classroom instruction only labs)]
  - CSC Labs [RVR 1013/2005/2009/2013/5029]
  - ECS Windows Terminal Server (Hydra)

CN1 Hello World App

- Steps for Eclipse:
  - File → New → Project → Codename One Project
  - Give a project name “HiWorldPrj” and **uncheck “Java 8 project”**. Hit “Next”.
  - Give a main class name “HiWorld”, package name “com.mycompany.hi”, and select a “native” theme, and **“Hello World(Bare Bones)” template (for manual GUI building)**. Hit “Finish”.
- It generates and builds the project. You can view your main class under the package explorer:

HiWorldPrj → src → com.myCompany.hi → HiWorld.java

CN1 Hello World App (cont.)

- Run the app on the simulator in Eclipse by right clicking the last entry of the project under the package explorer:

  HiWorldPrj → Simulator_HiWorldPrj.launch
  - Select “Run As” to run and “Debug As” to debug your app.
- You can also run it directly from the command-line. Get into the HiWorldPrj directory and (in Windows) type:

```
java -cp dist\HiWorldPrj.jar;JavaSE.jar
com.codename1.impl.javase.Simulator
com.mycompany.hi.HiWorld
```

(all in one line, but put spaces between sub-lines)

CN1 Hello World App (cont.)

- Unix-like operating systems (such as Linux and Mac OS X) use “forward-slash” and “colon” (instead of “back-slash” and “semicolon”:

  java -cp dist/HiWorldPrj.jar:JavaSE.jar
  com.codename1.impl.javase.Simulator
  com.mycompany.hi.HiWorld **(all in one line, but put spaces between sub-lines)**

- You can switch through different skins in the simulator. For assignments we will use iPad skin (download it via “Skins” → “More” -> “iPad III iOS 7”, to fit the skin to your screen uncheck “Scrollable” under “Simulate” menu)

Troubleshooting Problems

- If dist\HiWorldPrj.jar is not generated:

Right click on project and hit “Codename One → Send to Android Build”, then hit “Cancel”, if still does not work:

set **JAVA_HOME** environment variable to JDK directory

In Windows: goto “Control Panel → System → Advance System Settings -> Environment Variables” and add JAVA_HOME as *C:\Program Files\Java\jdk1.8.0_144* to “System Variables” (gray numbers indicate the latest release of Java)

- If the command line complains that:

  ‘java’ is not recognized … : add *C:\Program Files\Java\jdk1.8.0_144\bin* to **PATH**

  JavaSE.jar cannot be found … : (first make sure you are in the project directory that has JavaSE.jar) add current directory (indicated by a single period “.”) to **CLASSPATH**

(see Appendices.pdf for tips)

CN1 and Assignments

- For each assignment create a different CN1 project.
- You **must** create all assignments in the same way as HiWorldPrj example:
  - uncheck “Java 8 project”, select “native” theme, and **“Hello World (Bare Bones)”** template.
  - change the project, main class, and package names…

CN1 and Assignments (cont.)

- For instance for Assignment#1:
  - Project Name: A1Prj
  - Main Class Name: Starter (keep the same for all assignments)
  - Package: com.mycompany.a1
- Main class has the following structure:

```
public class Starter {
   …
   public void init(…) {…}
   public void start() {…}
   public void stop() {…}
   public void destroy() {}
}
```

CN1 and Assignments (cont.)

- Solve the assignment by modifying **start()** in Starter.java (**do NOT delete other methods**) and adding more java files (right click on the package then hit "New" → "Class").
- **Make sure** dist\A1Prj.jar is up to date (if not, in Eclipse, right click on dist directory and hit "Refresh" or right click on project and hit "Codename One → Send to Android Build", then hit "Cancel")
- **Make sure** your submission works from command-line. Go into the A1Prj directory and type:

java -cp dist\A1Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a1.Starter **(all in one line, but put spaces between sub-lines)**

CN1 and Assignments (cont.)

- Again, for Unix-like operating systems (such as Linux and Mac OS X) use "forward-slash" and "colon":

java -cp dist/A1Prj.jar:JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a1.Starter **(all in one line, but put spaces between sub-lines)**

- Deliverables:

Zip A1Prj.jar (under *dist* dir) and entire *src* dir to a file called YourLastName-YourFirstName-a1.zip & submit this zip file to SacCT.

Assignment#0

- Find a lab computer that has CN1 or install CN1 to your computer.
- Following the instruction in the previous slides, generate an empty project called A0Prj.
- Modify Starter.java by replacing the texts “Hi World” with “Assignment#0”. Run the simulator.
- Experiments with debugging options of your IDE.
- Verify that your submission also works from the command line.

Do <u>not</u> submit A0 via SacCT (its purpose is to make sure you have access to CN1 and ready to solve real assignments)

CN1 Online resources

- Developers guide:

CN1 Developer Guide - Revision 3.6 (pdf is available at SacCT)

- Video tutorials can be found at:

http://www.codenameone.com/how-do-i.html

(note: mostly give examples that use the GUI builder which we will not utilize)

- You can view JavaDocs of APIs:

https://www.codenameone.com/javadoc/index.html

3 - OOP Concepts

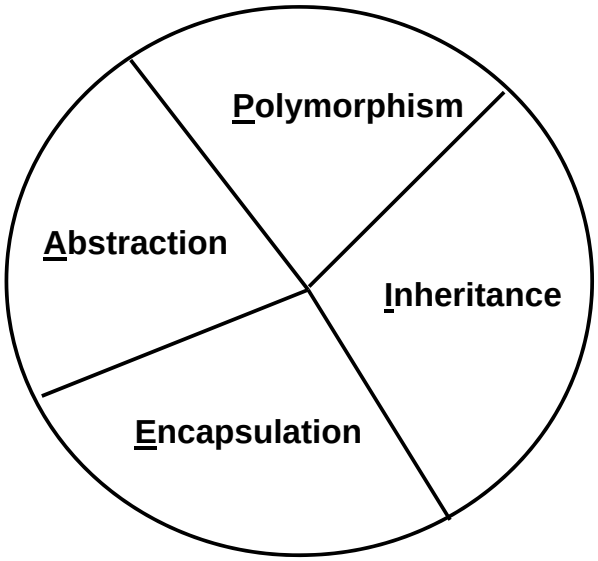
Computer Science Department
California State University, Sacramento

Overview

- The OOP “A PIE”
- Abstraction
- Encapsulation: Bundling, Information Hiding, Implementing Encapsulation, Accessors & Visibility
- UML Class Diagrams
- Class Associations: Aggregation, Composition, Dependency, Implementing Associations

The OOP "A Pie"

- Four distinct OOP Concepts make "A PIE"



Abstraction

- Identification of the minimum essential characteristics of an entity
- Essential for specifying (and simplifying) large, complex systems
- OOP supports:
  - *Procedural* abstraction
  - *Data* abstraction

(clients do not need to know about implementation details of identified procedures and data types, e.g. Stack)

Encapsulation

In Java encapsulation is done via classes.

"Bundling"

- Collecting together the data and procedures associated with an abstraction
- Class has fields (data) and methods (procedures)

"Information Hiding"

- Prevents certain aspects of the abstraction from being accessible to its clients
- Visibility modifiers: public vs. protected vs. private
- Correct way: keep all data **private** and use accessors (Getters/Selectors vs. Setters/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

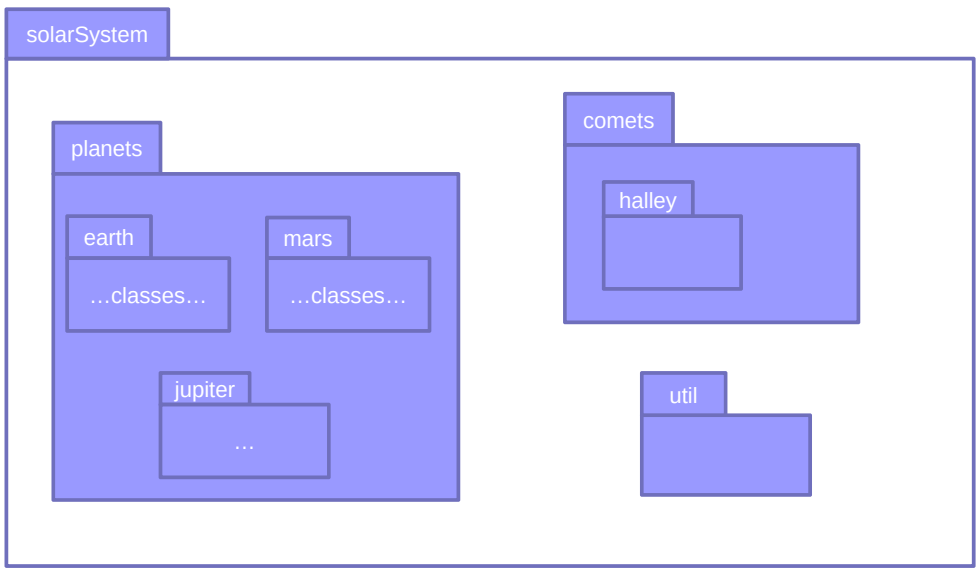
Access (Visibility) Modifiers

	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



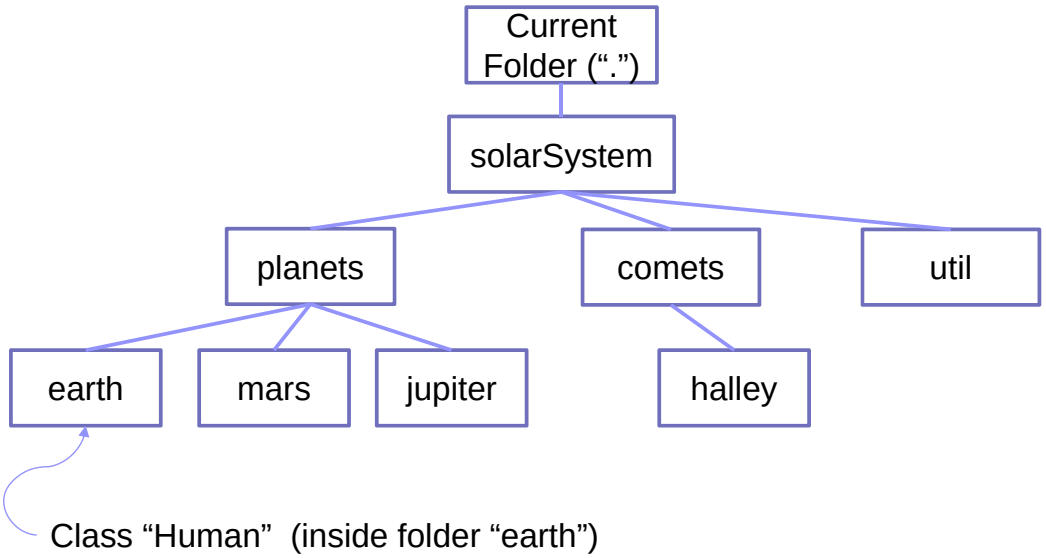
Java Packages (cont.)

- 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:



Abstraction example: Color

- We see colors at the visible portion of the electromagnetic spectrum.
  - Color can be represented by its wavelength.
  - Better approach: use abstraction and represent them with a color model (RGB, CMYK).
- Three axes: Red, Green, Blue
- Distance along axis = intensity (0 to 255)
- Locations within cube = different colors
  - Values of equal RGB intensity are grey

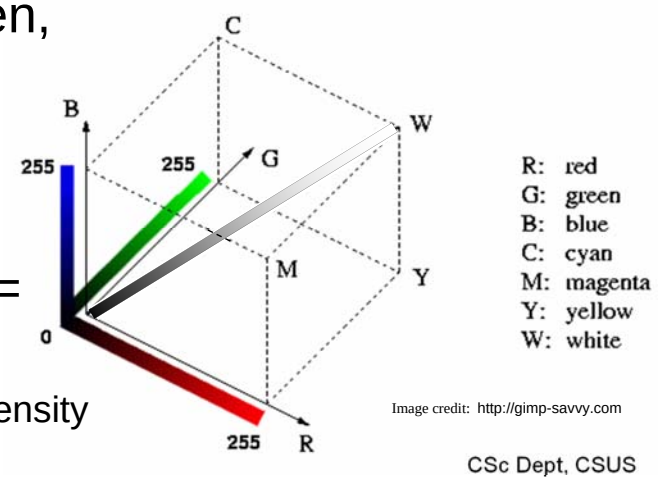


Image credit: http://gimp-savvy.com

Example: CN1 `ColorUtil` Class

- An *encapsulated abstraction*
- Uses “RGB color model”
- **`ColorUtil`** is in:
  - **`com.codename1.charts.util`**
- Has static functions to set color and get color, and static *constants* for many colors:

```
import com.codename1.charts.util.ColorUtil;
int myColor = ColorUtil.rgb(255 , 255, 255);  //set color to white
myColor = ColorUtil.rgb(255, 0, 0);           //change the color to red
myColor = ColorUtil.BLACK;                    //same as ColorUtil.rgb(0 , 0, 0)
myColor = ColorUtil.GREEN;                    //same as ColorUtil.rgb(0 , 255, 0)
System.out.println ("myColor: " + "[" + ColorUtil.red(myColor) + "," +
                                        ColorUtil.green(myColor) + "," +
                                        ColorUtil.blue(myColor) + "]";
                                        //prints: myColor = [0, 255, 0]
```

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.)

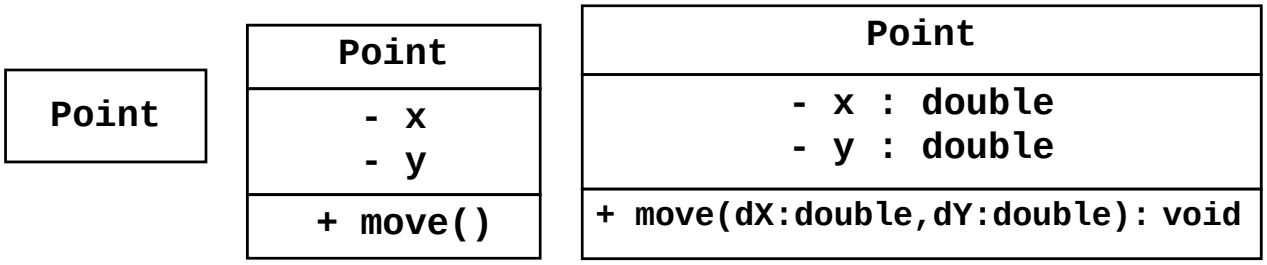
- **The correct way, with "Accessors":**

```
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

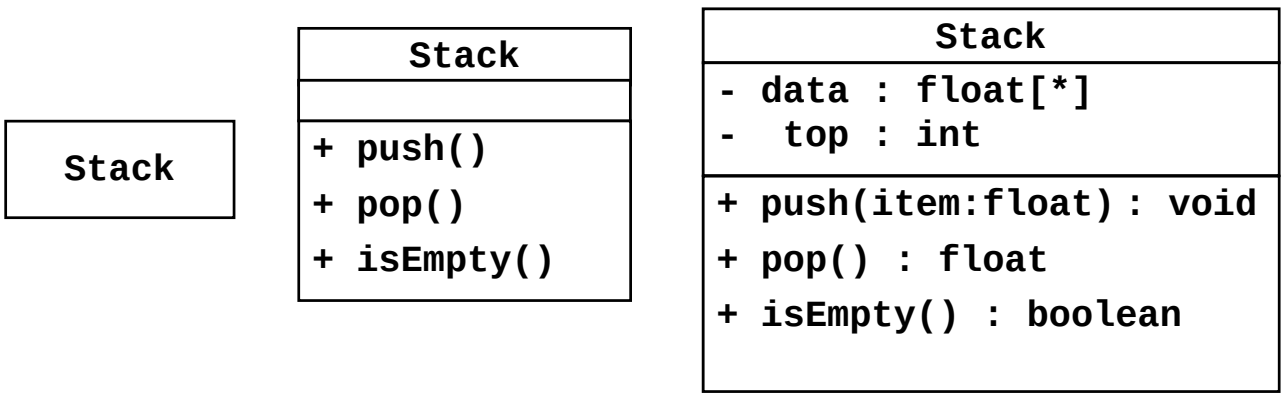
UML "Class Diagrams"

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



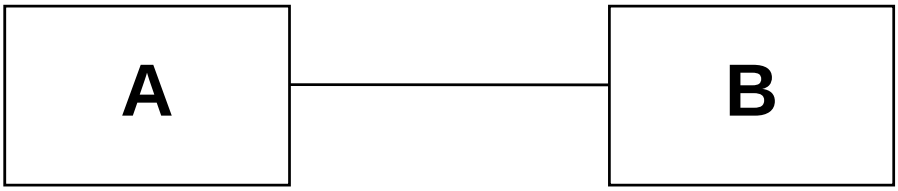
UML "Class Diagrams" (cont.)

  - UML for the "**`Stack`**" class:



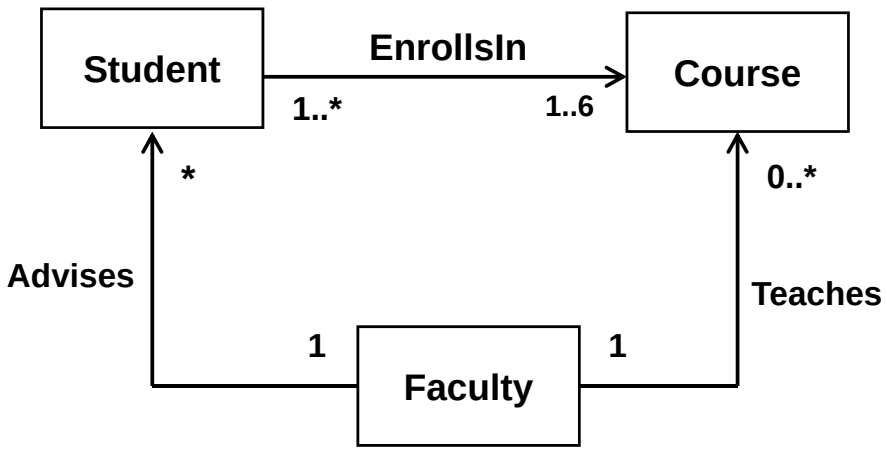
Associations

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



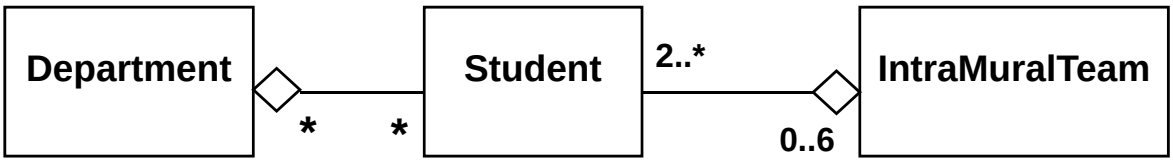
Associations (cont.)

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



Special Kinds Of Associations

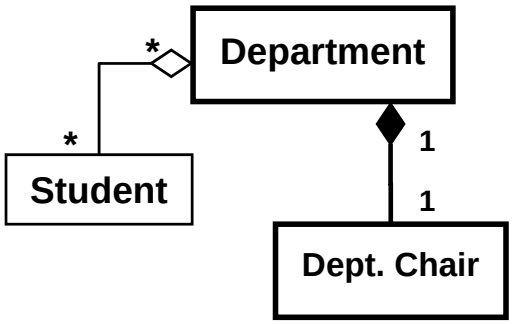
- **Aggregation**
  - Represents ***"has-a"*** or ***"is-Part-Of"***



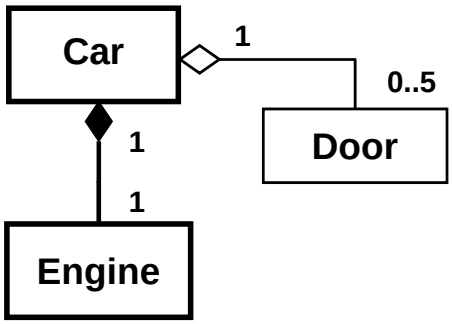
- **An IntraMuralTeam is an aggregate of *(has)* 2 or more Students**
- **A Student *is-a-part-of* at most six Teams**
- **A Department has any number of Students**
- **A Student can belong to any number of Departments (e.g. double major)**

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)



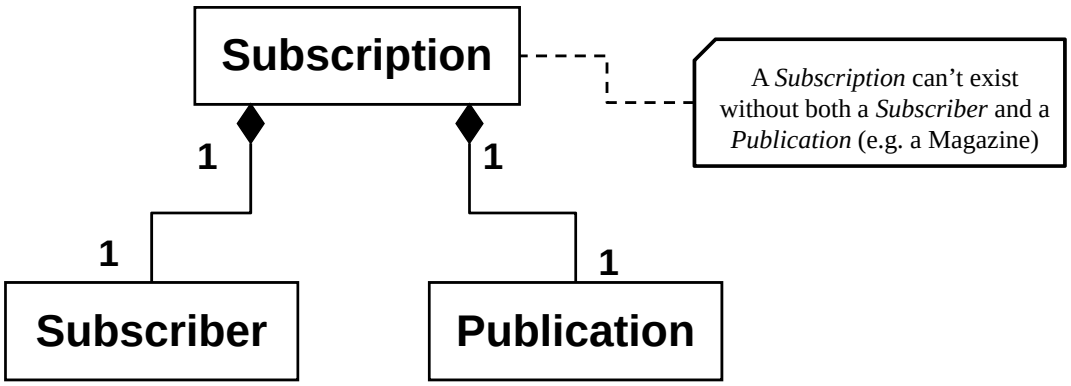
Exclusive ownership



Required ownership

Special Kinds Of Associations (cont.)

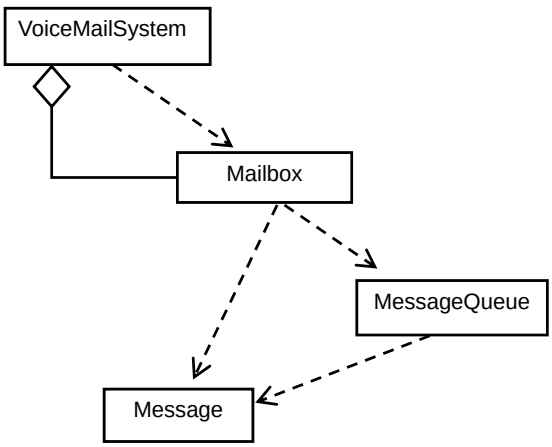
- Composition (another example)



Special Kinds Of Associations (cont.)

- Dependency
  - **Represents "uses" (or "knows about")**

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

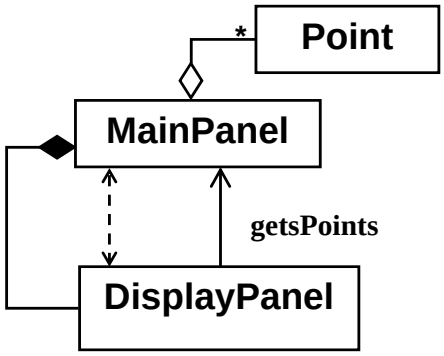


Implementing Associations

- 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
   public 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);
        }
    }
}
```

4 - Inheritance

Computer Science Department
California State University, Sacramento

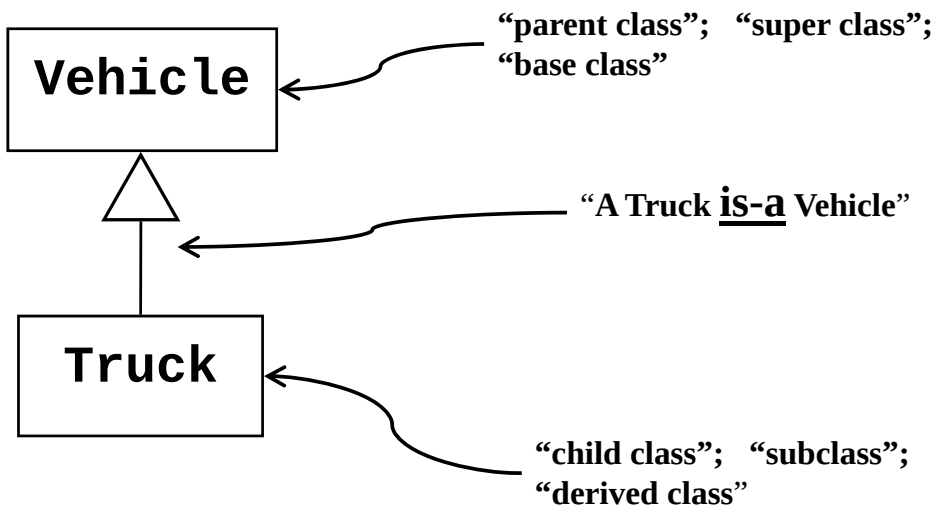
Overview

- **Definition**
- **Representation in UML, Implementation in Java, The "IS-A" concept**
- **Inheritance Hierarchies**
- **Overriding, Overloading**
- **Implications for Public vs. Private data**
- **Forms of Inheritance: Extension, Specialization, Specification**
- **Abstract classes and methods**
- **Single vs. Multiple Inheritance**

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



Inheritance In Java

- **Specified with the keyword "extends" :**

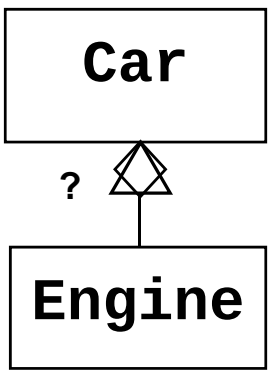
```
public class Vehicle {

  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 single "extends" allowed (no "multiple inheritance")**
- **Absence of any "extends" clause implies "extends Object"**

The "IS-A" Relationship

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



An Engine "is a" Car ? X

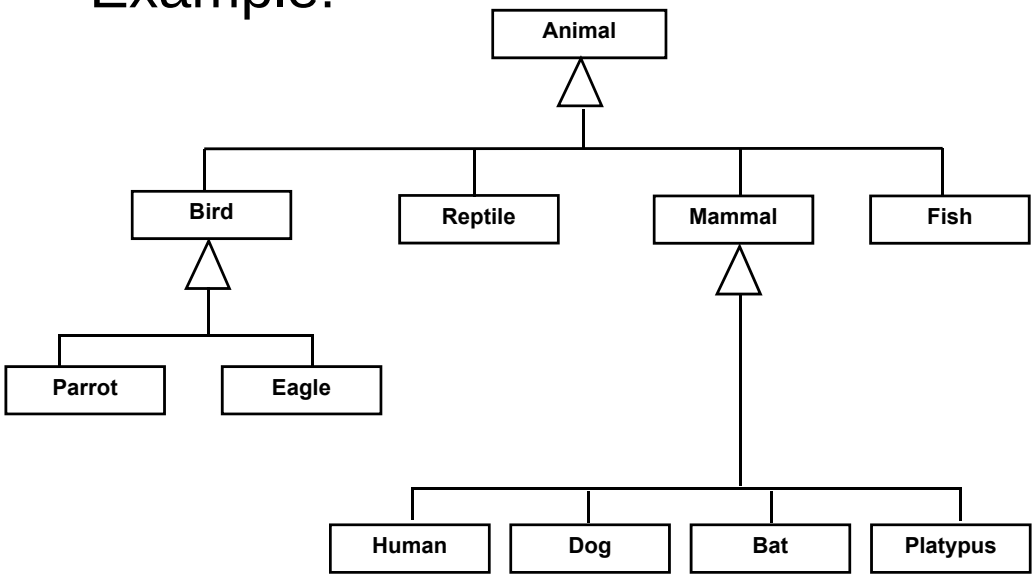
A Car "is an" Engine ? X

A Car "has-an" Engine ✓

An Engine "is a part of" a Car ✓

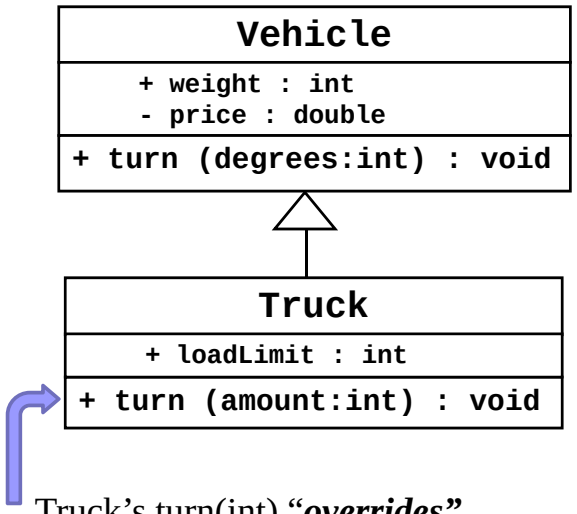
Inheritance Hierarchies

- Example:



Method Overriding

- Inheritance leads to an interesting possibility:
  duplicate method declarations



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

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

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

  ...
}
```

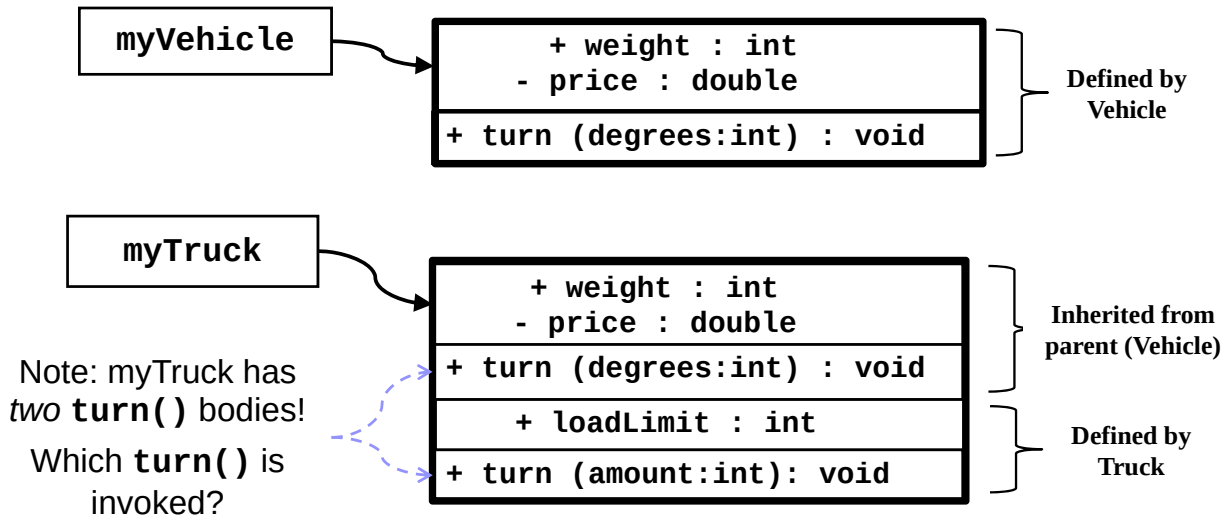
```
public class Truck extends Vehicle {
  public int loadLimit ;

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

  ...
}
```

Effects of Method Overriding

Consider the following code:

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

… then we get two objects:



Method Overriding: Summary

- **Occurs when a child class redefines an inherited method, which:**
  - **has same name**
  - **has same parameters**
  - **returns same type or subtype**
- **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 which is not a subtype*.**
  - **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 parameter types**
  - **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:

```
Circle(); Circle(Color c); Circle(int radius);
 Circle(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

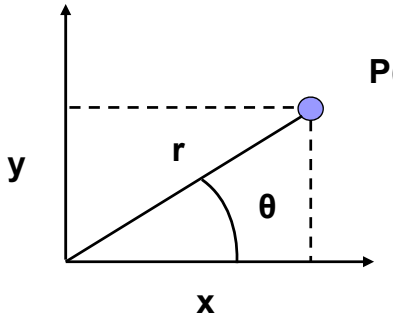
Now we will learn why!

`Point` (with "Accessors"):

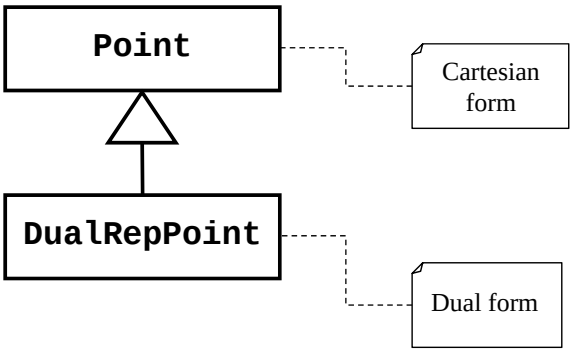
```
public class Point {
   private double x, y ;
   public Point (){
      x = 0.0 ;
      y = 0.0 ;
   }
   public double getX() {
      return x ;
   }
   public double getY() {
      return y ;
   }
   public void setX (double newX) {
      x = newX ;
   }
   public void setY (double newY) {
   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

GOOD

Example: extend "**Point**" to create "**DualRepPoint**"



P(x,y) = P(r,θ)



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 = 2.0 ;
    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 ;                    ← New: private access

  // 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: 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

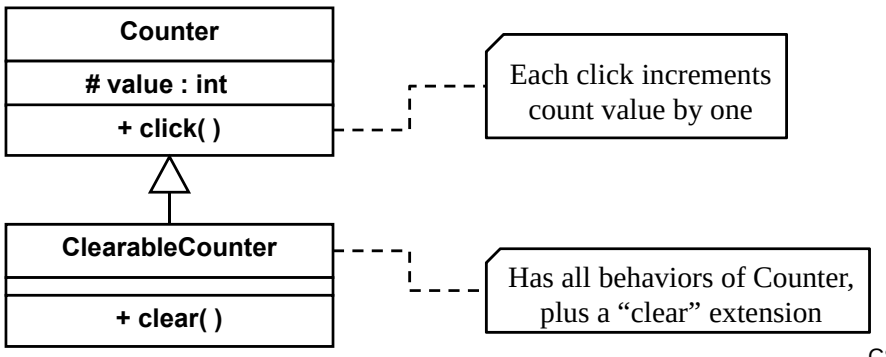
```
public class DualRepPoint extends Point  {   //uses "Good" Point with accessors
  private double radius, angle ;
  //...constructors and accessors for radius and angle here as before ...
  // Override inherited accessors
  public void setX (double xVal) {  //note that overriding the parent accessors
    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() ;      // 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)**

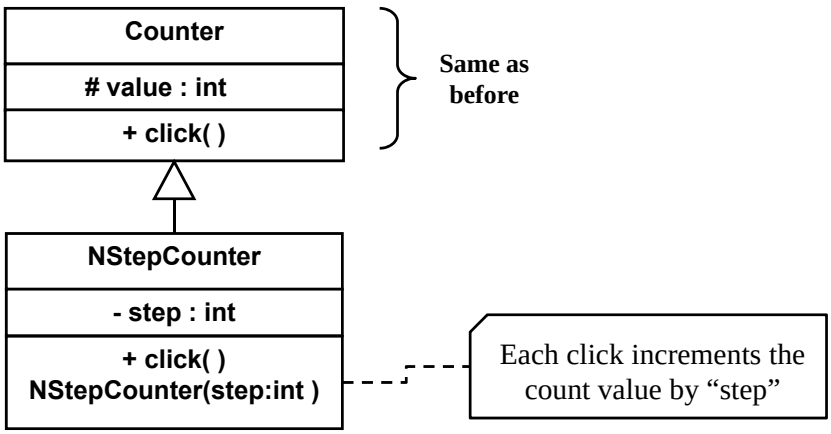


Inheritance for Extension (cont.)

```
/** 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)**
- **Uses *overriding* to change the behavior**
- **Example: N-Step Counter**

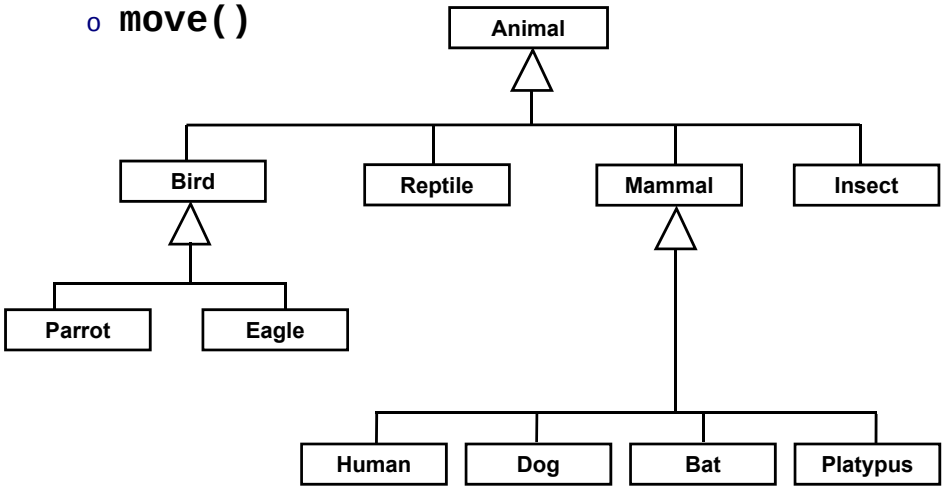


Inheritance for Specification

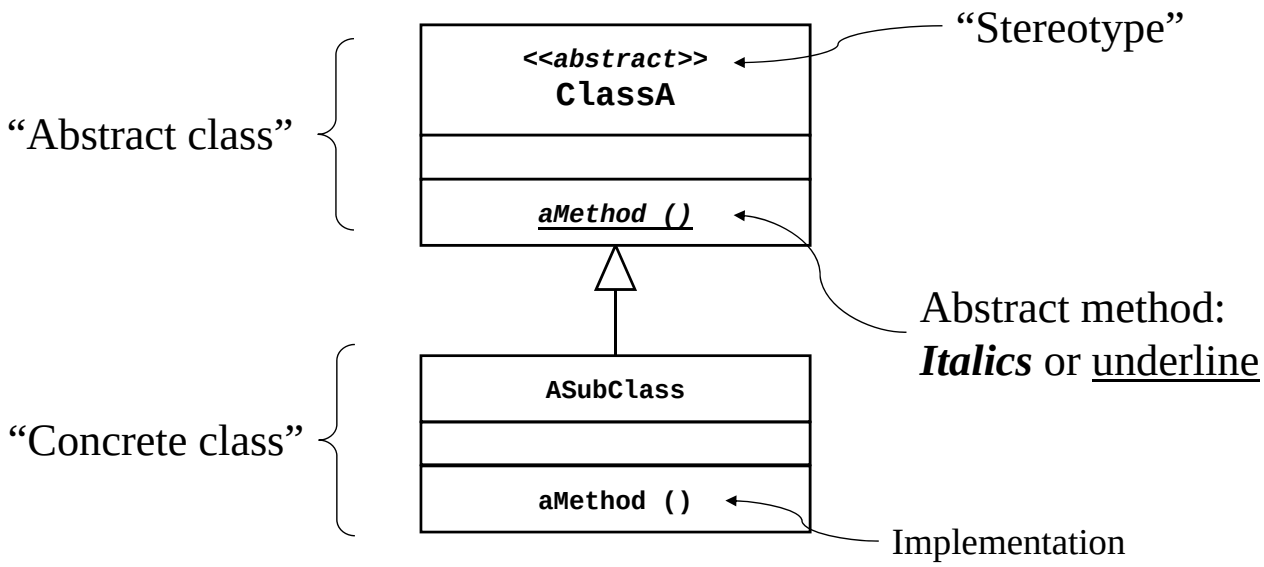
- **Used to *specify (define)* behavior *declared* (but not *defined*) by the parent**
  - **Classes which declare but don't define behavior: Abstract Classes**
  - **Methods which don't contain implementations: *Abstract methods***

Abstract Classes & Methods

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

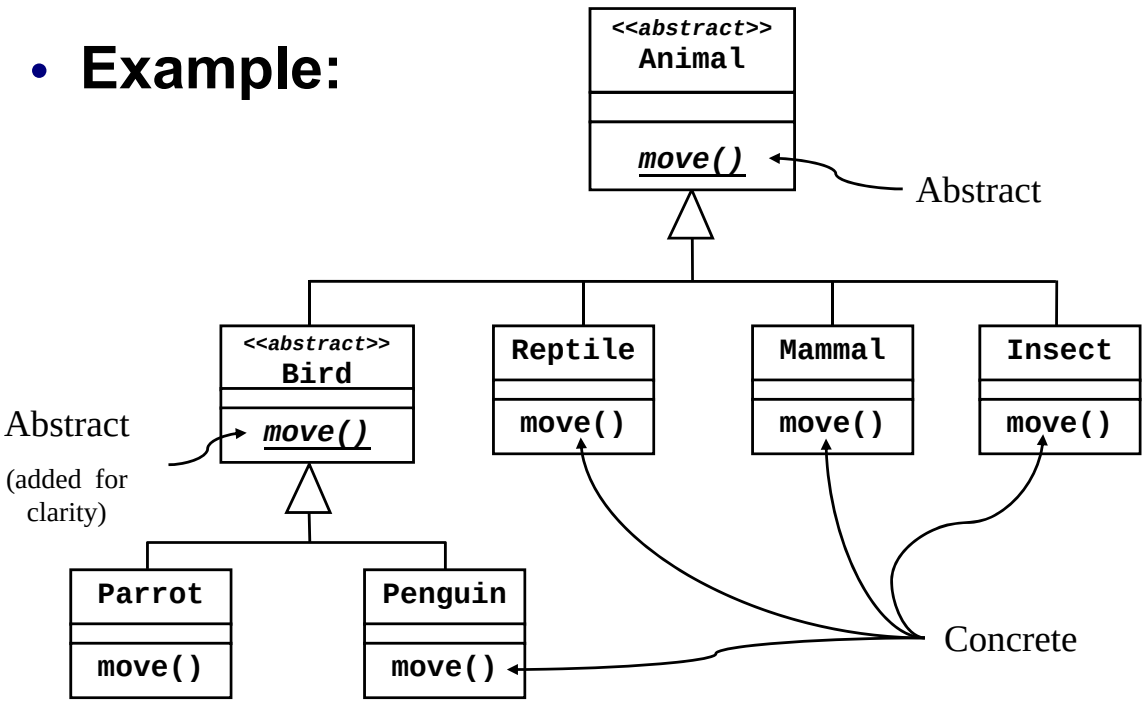


Inheritance for Specification (cont.)



Inheritance for Specification (cont.)

- **Example:**

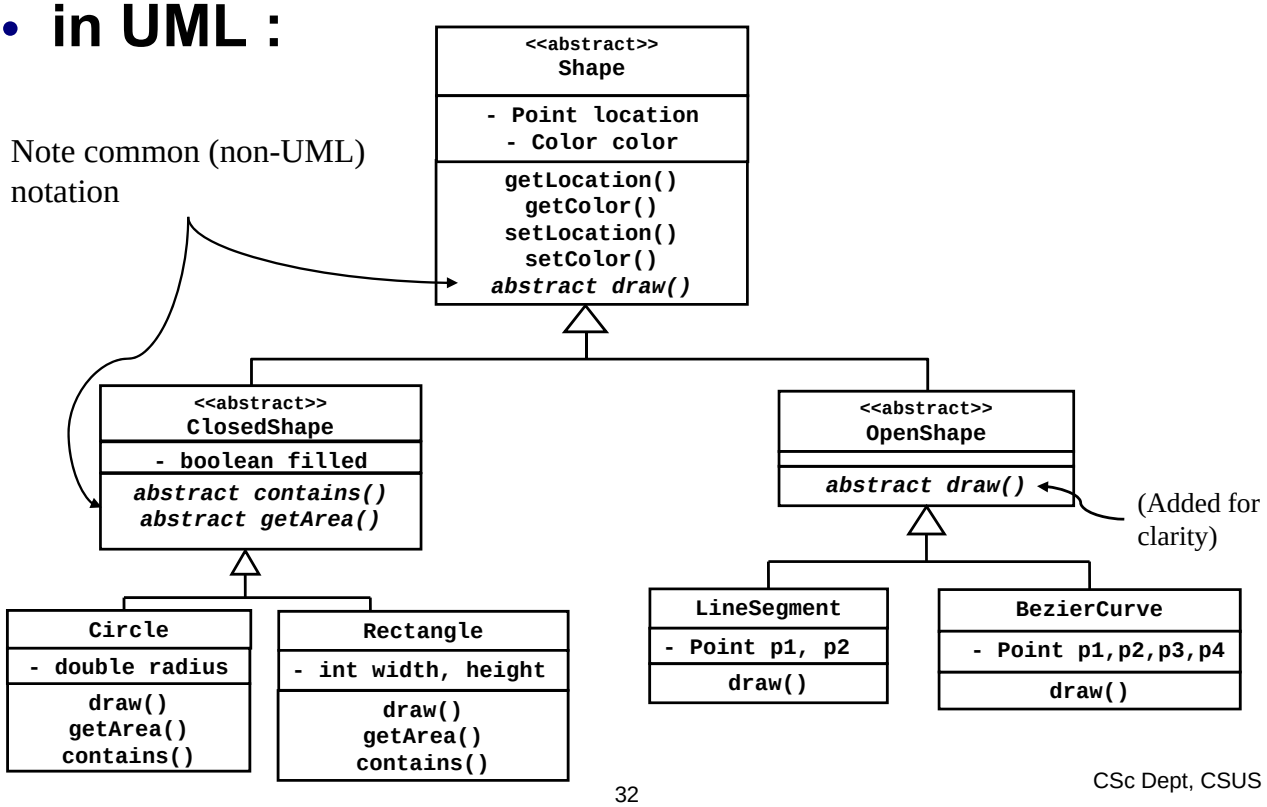


Inheritance for Specification (cont.)

- **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 :**



Java Abstract Classes

- Both classes and methods can be declared abstract

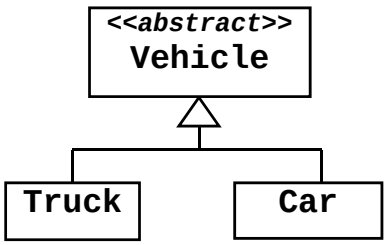
```
public abstract class Animal {
      public abstract void move () ;
}
```

- 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 int color;
  private Point location;

  public Shape() {
    color = ColorUtil.rgb(0,0,0);
    location = new Point (0,0);
  }

  public Point getLocation() {
    return location;
  }

  public int getColor() {
    return color;
  }

  public void setLocation (Point newLoc) {
    location = newLoc;
  }

  public void setColor (int newColor) {
    color = newColor;
  }

  public abstract void draw(Graphics g);
}
```

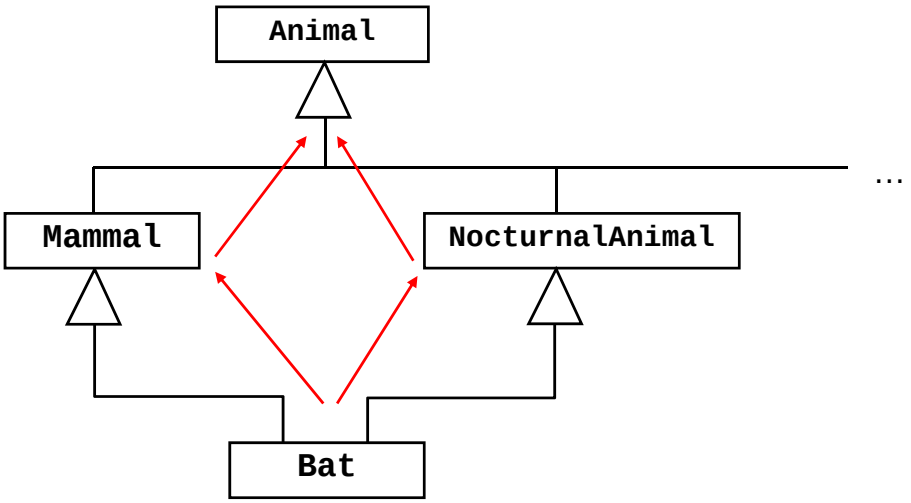
Example: Abstract Shapes (cont.)

```
/** 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() {
    //automatically calls super() – no-arg constructor of its parent (Shape)
    filled = false;
  }
  public ClosedShape(boolean filled) {
    //automatically calls super() – no-arg constructor of its parent (Shape)
    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);  //no-arg constructor of its parent (ClosedShape) is not called
    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();
}
```

V - Polymorphism

Computer Science Department
California State University, Sacramento

Overview

- **Definitions**
- **Static ("compile-time") Polymorphism**
- **Polymorphic references, Upcasting / Downcasting**
- **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 on various types of objects
  - An operation that can be done in a variety of ways
  - A reference can be assigned to different types

"Static" Polymorphism

Detectable *during compilation.*

Example: Operator overloading:

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

- The "+" can perform on *different* types of objects
- "+" can therefore be thought of as a "*polymorphic operator*"

"Static" Polymorphism (cont.)

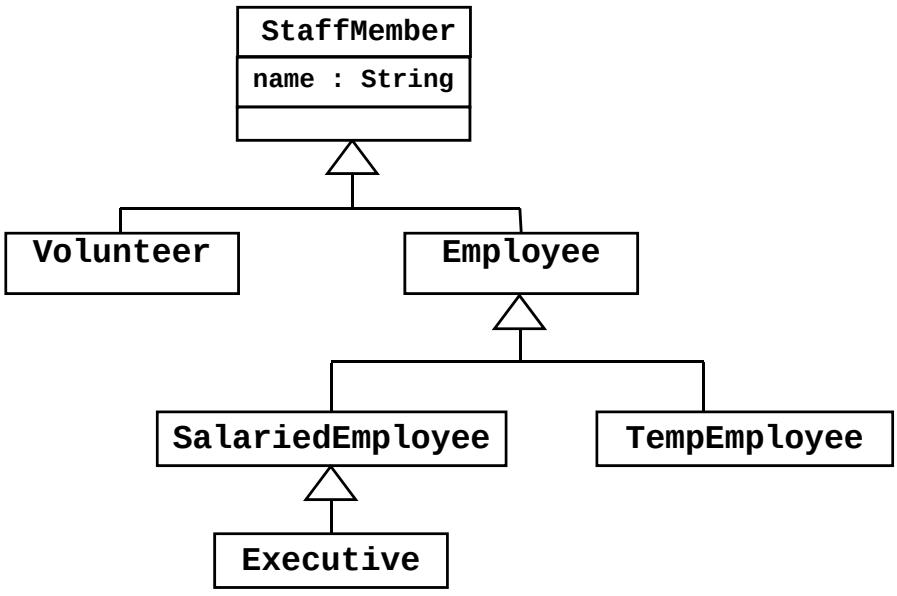
Another example: Method overloading:

```
//return the distance to an origin
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

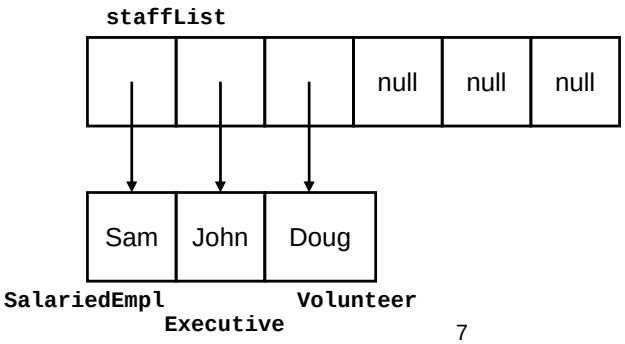
Consider the following class hierarchy:



Polymorphic References (cont.)

- A "polymorphic reference" can *refer to different object types at runtime*:

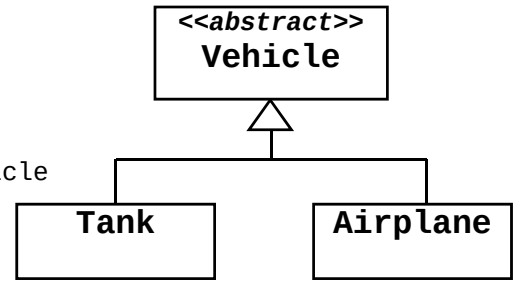
```
StaffMember [ ] staffList = new StaffMember[6];
. . .
staffList[0] = new SalariedEmployee ("Sam");
staffList[1] = new Executive ("John");
staffList[2] = new Volunteer ("Doug");
. . .
```



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
```

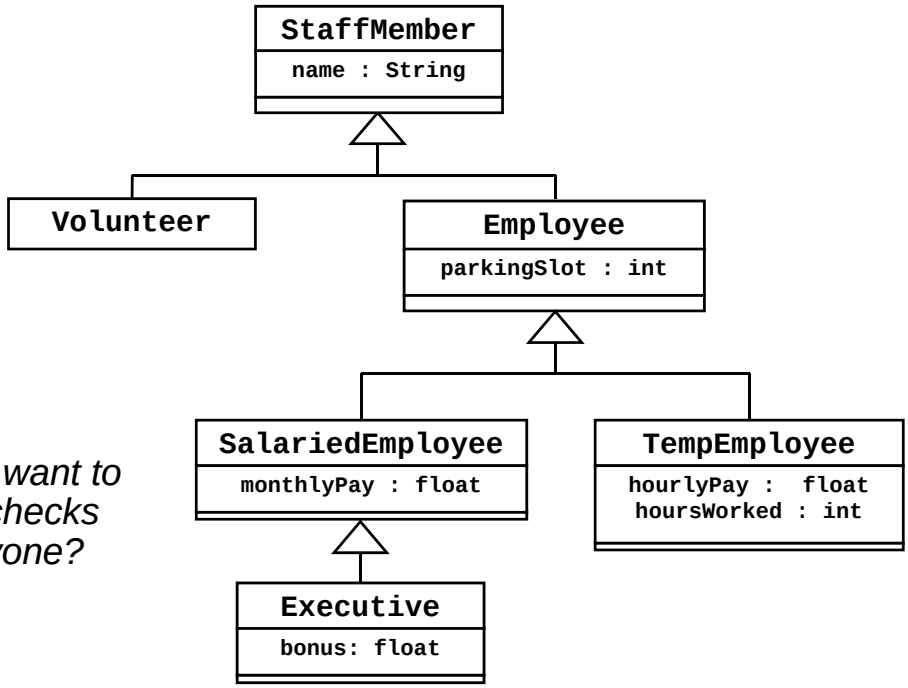


- "Downcasting" requires casting:

```
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?

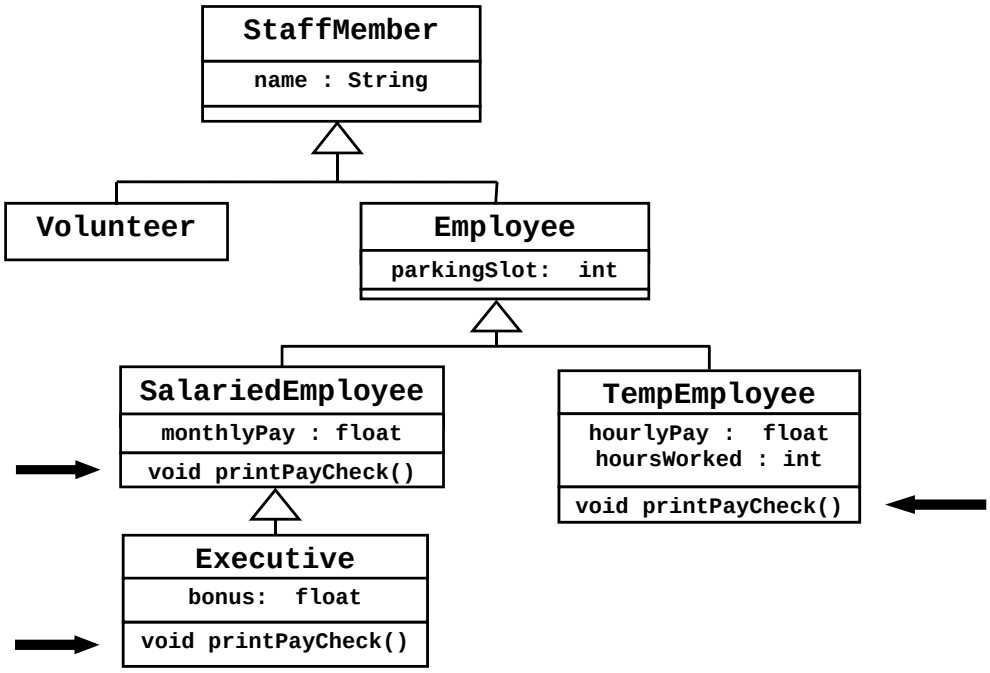
Runtime Polymorphism (cont.)

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() ;
}
...
```

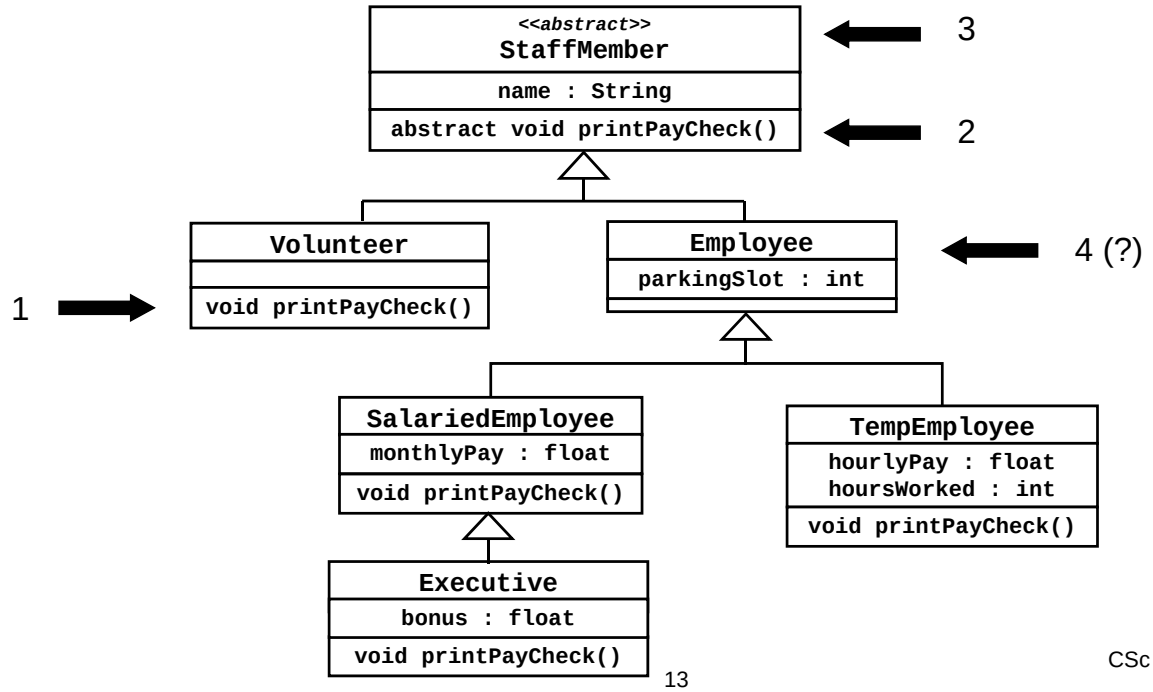
Now, the Print method which gets invoked is:

- *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

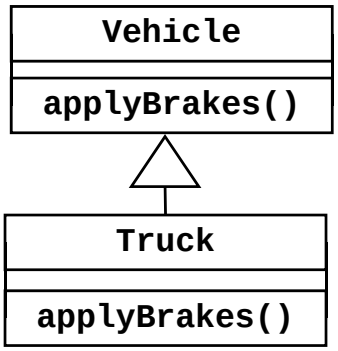
Ideally, *every* class should know how to deal with "**`printPayCheck`**" messages:



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 "**`static`**, **`final`**, and/or **`private`**"
- **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...
```

6 - Interfaces

Computer Science Department
California State University, Sacramento

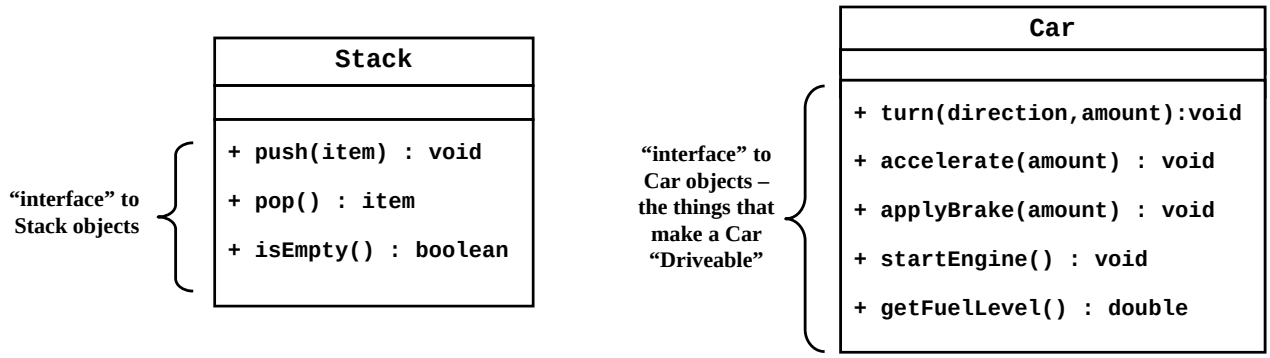
Overview

- **Class Interfaces, UML Interface Notation, The Java *Interface* Construct**
- **Interfaces in C++**
- **Predefined Interfaces**
- **Interface Hierarchies**
- **Interface Subtypes**
- **Interfaces and Polymorphism**
- **Abstract Classes vs. Interfaces**
- **Multiple Inheritance via Interfaces**

CLASS INTERFACES

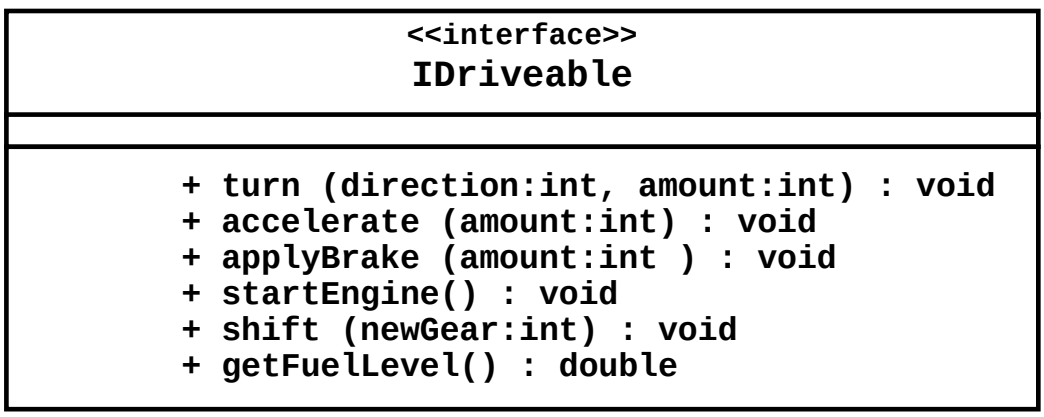
Every class definition creates an “interface”

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



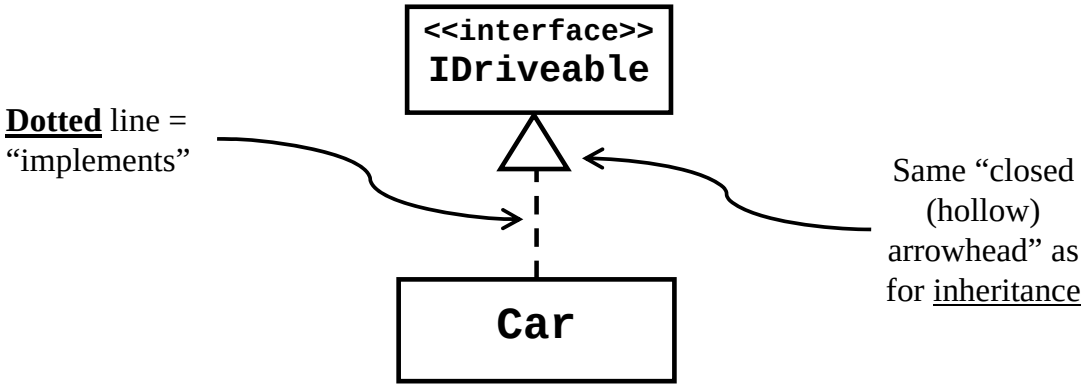
UML Interface Notation

or



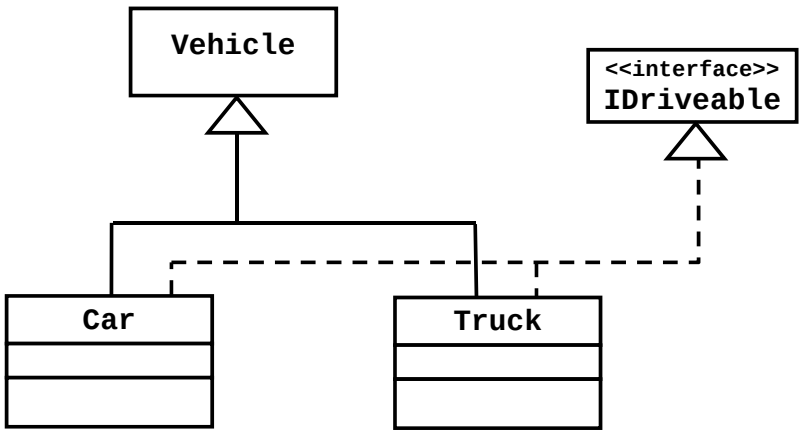
UML Interface Notation (cont.)

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



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
  - All methods are **public**
- Usually does *not* specify any *implementation* *(generally have abstract methods)*
  - *Java 8 introduced "default" and "static" interface methods that have body*
- Can have fields
  - All fields are **public** AND **static** AND **final**

(default visibility for interface fields and methods is public instead of package-private)

Java *Interface* construct (cont.)

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`**" in a concrete class == "provides bodies for all abstract 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 :**

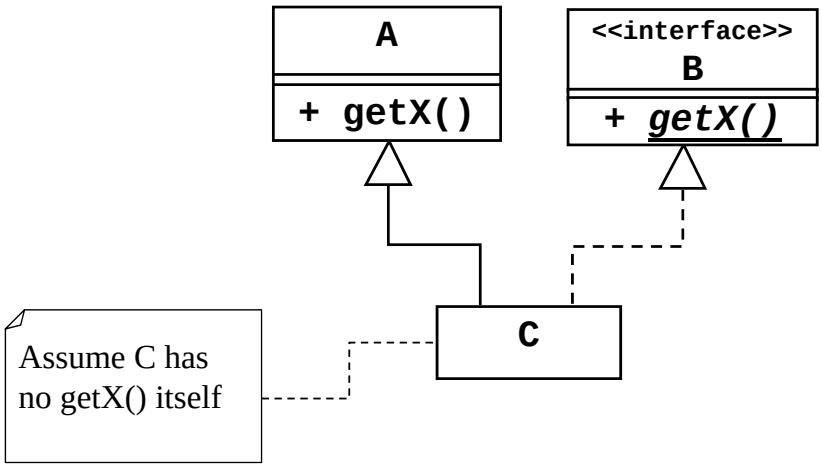
```
class IDriveable {
  public:
        virtual void turn (int direction, int amount) = 0 ;
        virtual void accelerate (int amount) = 0 ;
        ...
};
```

- **"Abstract" Classes as Interfaces :**

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

Quiz:

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



Predefined Interfaces in CN1

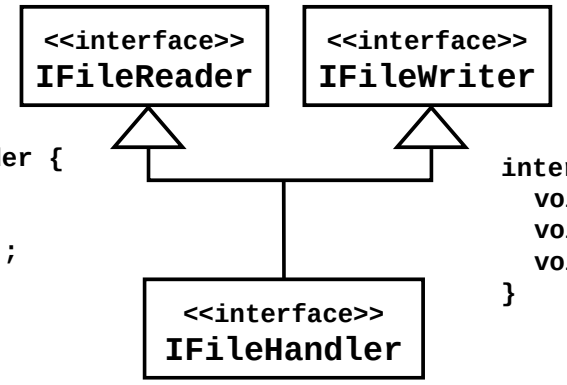
- **Many CN1 Classes implement built-in interfaces**
- **User Classes can also implement them**

Examples:

```
interface Shape {
  boolean contains(int x, int y);
  Rectangle getBounds();
  Shape intersection(Rectangle rect);
  //other methods...
}


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

Interface Hierarchies

Interfaces can *extend* other interfaces




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

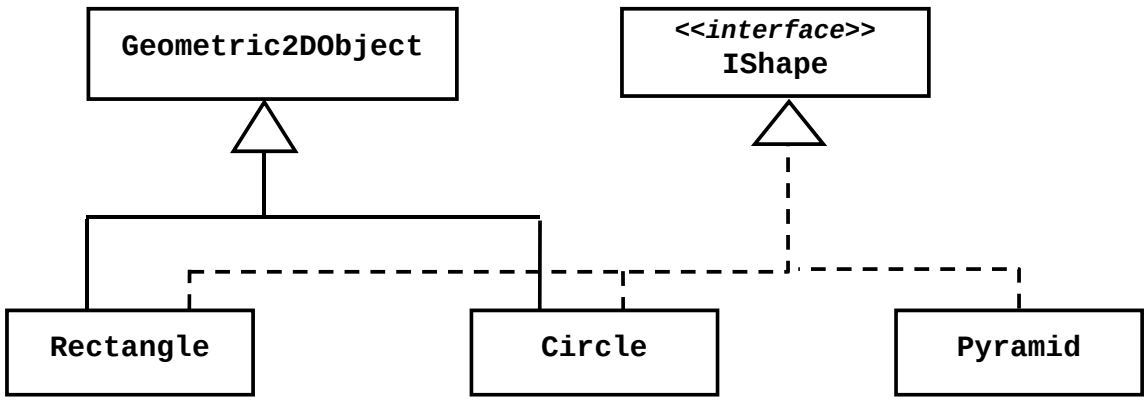
```
interface IFileWriter {
  void writeByte (byte b);
  void writeInt (int theInt);
  void writeString (String s);
}
```

```
interface IFileHandler extends IFileReader, IFileWriter {
  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" IShape**



Interface Subtypes (cont.)

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

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

- **Interfaces, like superclasses, provide objects with:**

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

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

Interfaces and Polymorphism

- *Apparent* type = What does it look like at a particular place in program (changes).

Determines: What methods may be invoked

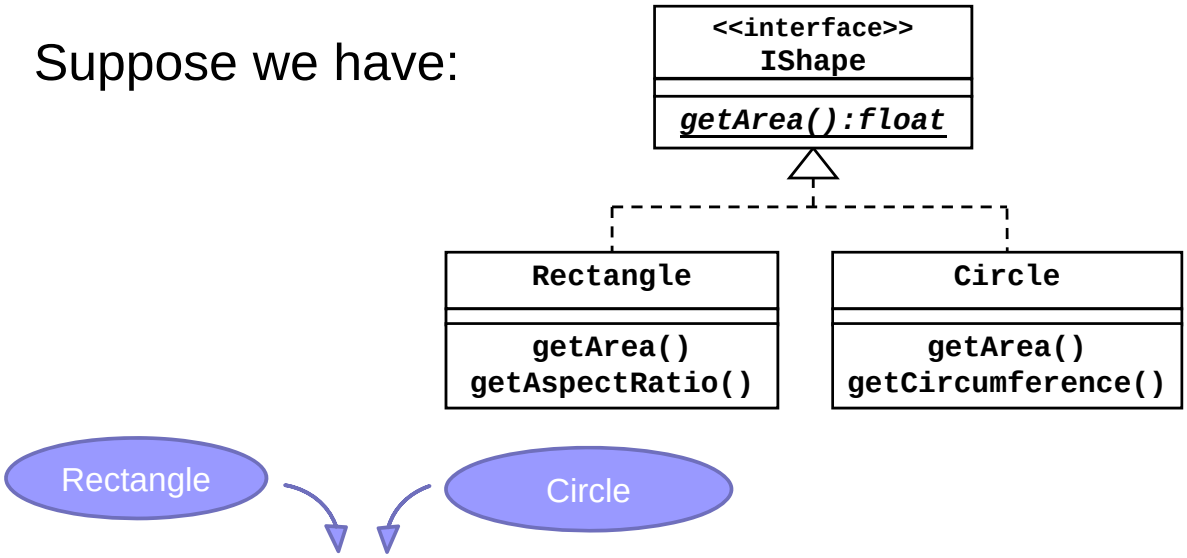
- *Actual* type = What was it created from (never changes)

Determines: Which implementation to call when the method is 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 IShape object, making calls to IShape methods.
  // Note this code only knows the apparent type, and only IShape methods
  // are visible – but any methods invoked are those of the actual type.
}
```

Interface Polymorphism Example

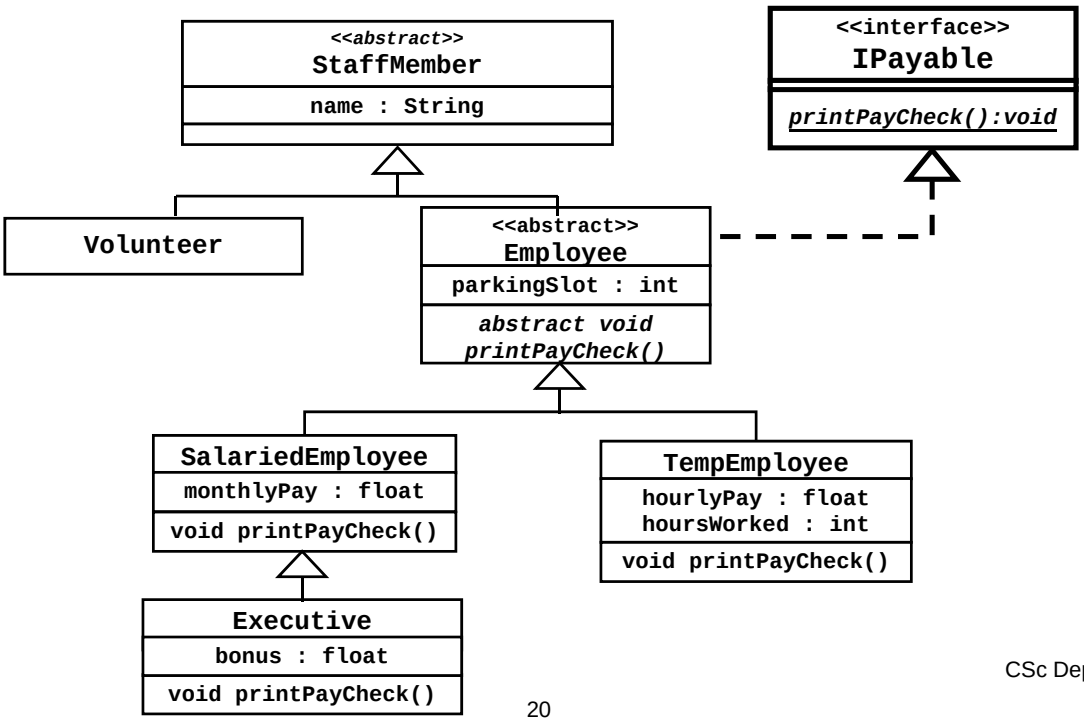
- Suppose we have:




```
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 abstract 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 illegal (in Java) to also 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.)

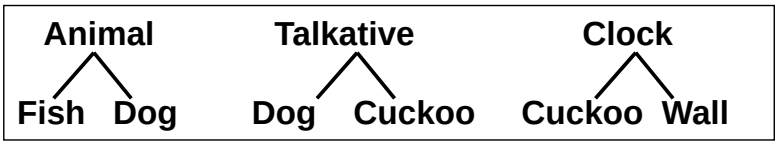
Use of interfaces can *increase Polymorphism:*

```
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”)
- We need non-public, non-static, or non-final fields OR private or protected methods
- Before Java 8:
  - There are at least *some* concrete methods shared between subclasses
  - We need to *add new methods* in the future (adding concrete methods to an abstract class does NOT break its subclasses)

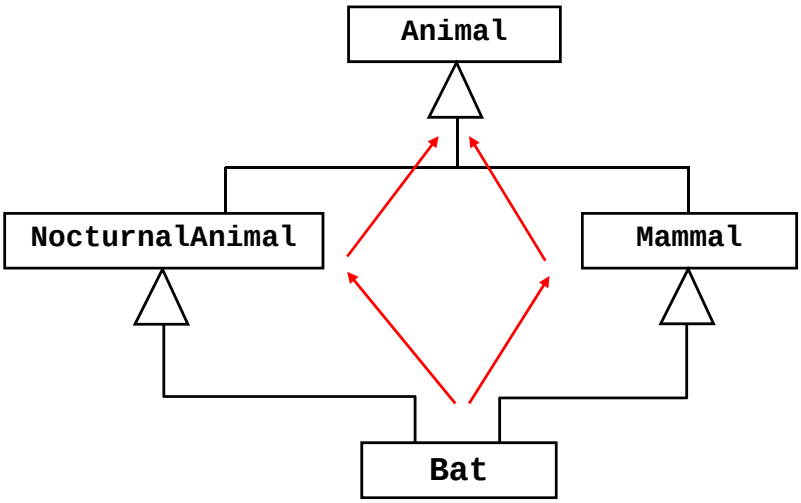
Abstract Class vs. Interface: Which?

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
- Before Java 8:
  - An API is likely to be stable (again: adding interface methods *breaks implementing classes)*
- Something like Multiple Inheritance is desired

(see next slides…)

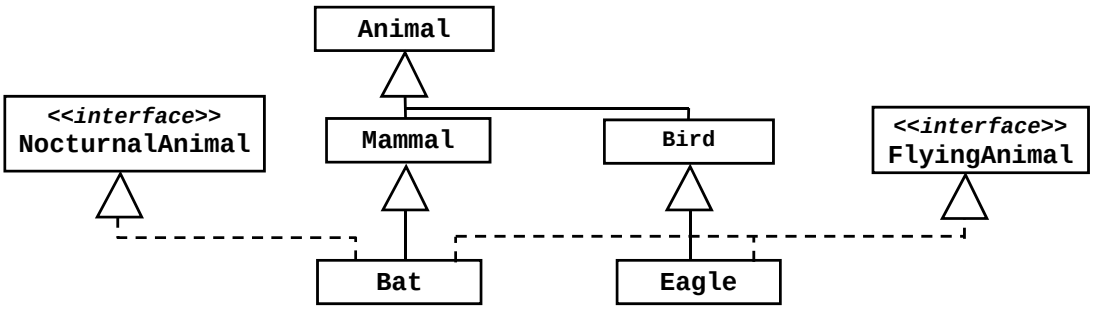
Multiple Inheritance Revisited



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 (before Java 8)**

7 - Design Patterns

Computer Science Department
California State University, Sacramento

Overview

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

Composite	***Singleton***
Iterator	***Observer***
Strategy	***Command***
Proxy	***Factory Method***

- **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:
  ring road, circular staircase 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:

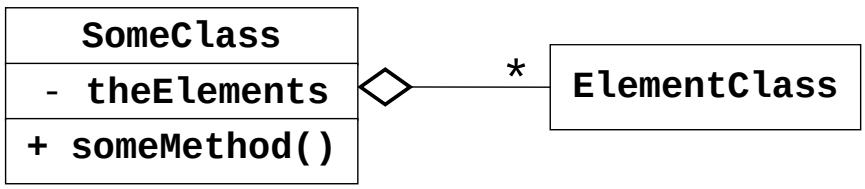
- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral:

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

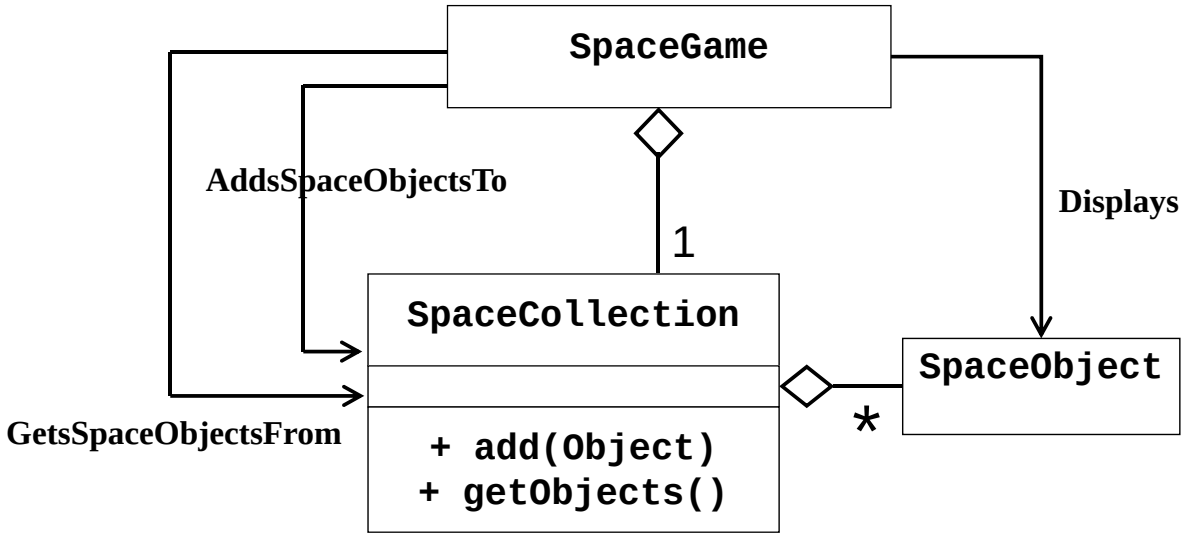
The Iterator Pattern

- 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* does not need know the data structure

Collection Classes



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 ICollection {
   public void add(Object newObject);
   public IIterator getIterator();
}


public interface IIterator {
   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 ICollection {
    private Vector theCollection ;
    public SpaceCollection() {
        theCollection = new Vector ( );
    }
    public void add(Object newObject) {
        theCollection.addElement(newObject);
    }
    public IIterator getIterator() {
        return new SpaceVectorIterator ( ) ;
    }
    ...continued...
```


SpaceCollection With Iterator (cont.)

```
private class SpaceVectorIterator implements IIterator {
    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
```

UML Notation for an Inner Class


```
public class A {
  private class B {
    ...
  }
}
```


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() {
        IIterator theElements = theSpaceCollection.getIterator() ;
        while ( theElements.hasNext() ) {
            SpaceObject spo = (SpaceObject) theElements.getNext() ;
            System.out.println ( spo ) ;
        }
    }
}
```

CN1'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. Can only be called once after **`next()`** was called. Optional operation. Exception is thrown if not supported or **`next()`** is not properly called.

CN1'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.

CN1'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());
      }
   }
}
```

(CN1 `Collection` interface is a subinterface of CN1 `Iterable` interface.)

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:**

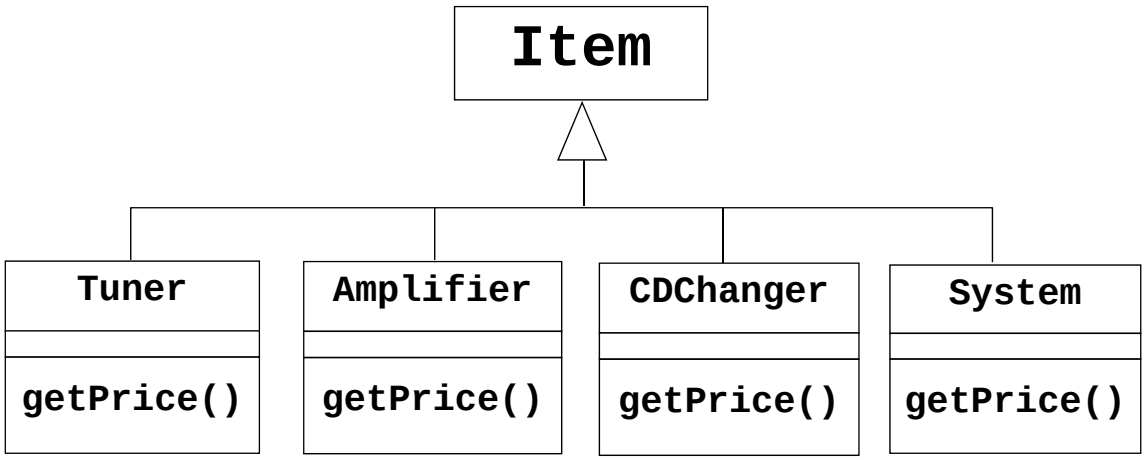
```
using namespace std;
vector<SpaceObject> myObjs;    //create a container
  //... code here to add SpaceObjects to the container (vector)
vector<SpaceObject>::iterator itr;    //declare an iterator
for (itr = myObjs.begin(); itr != myObjs.end(); itr++) {
   cout << *itr ;   //output next obj pointed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

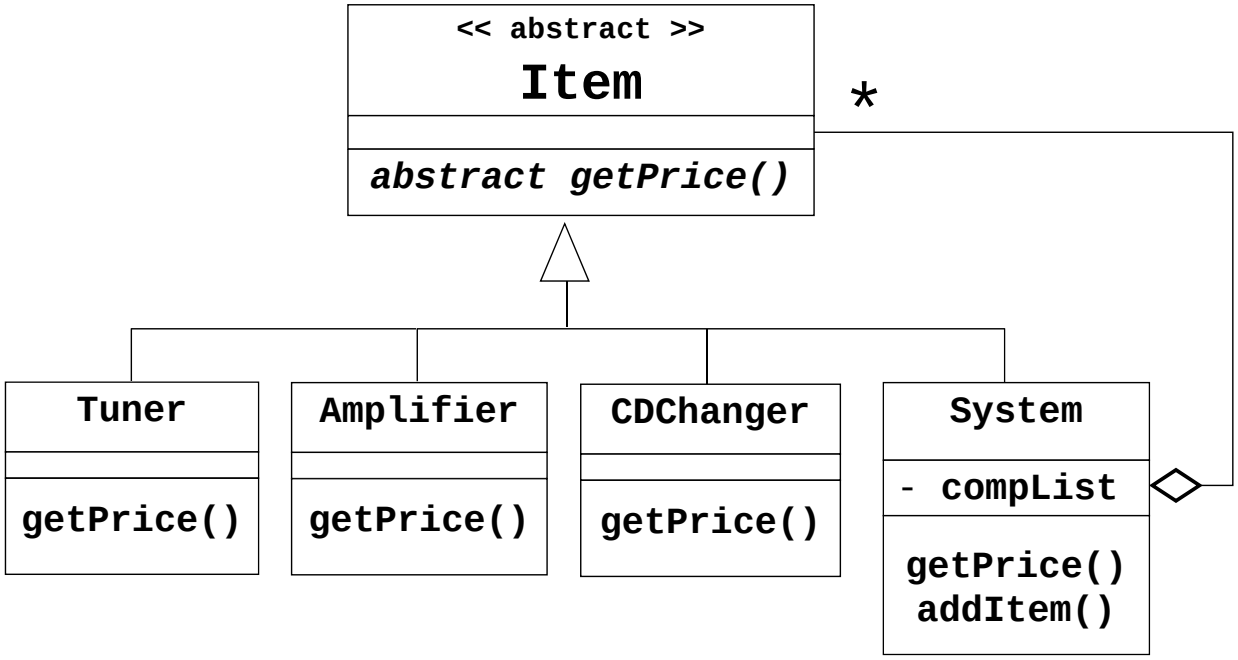
    Tuners, Amplifiers, CDChangers, etc.
    - Each item has a **`getPrice()`** method
  - The store also sells complete stereo systems
    - Systems also have a **`getPrice()`** method which returns a discounted sum of the prices.

Possible Class Organization

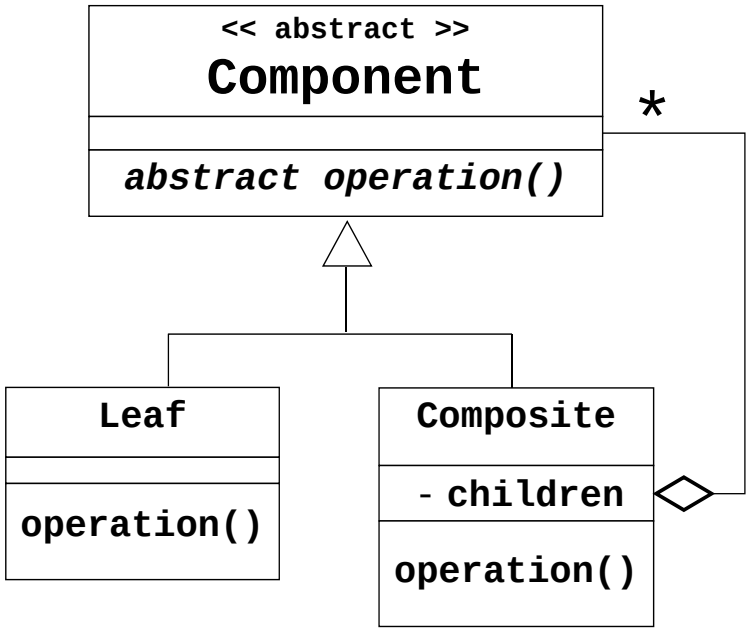


Problem ?

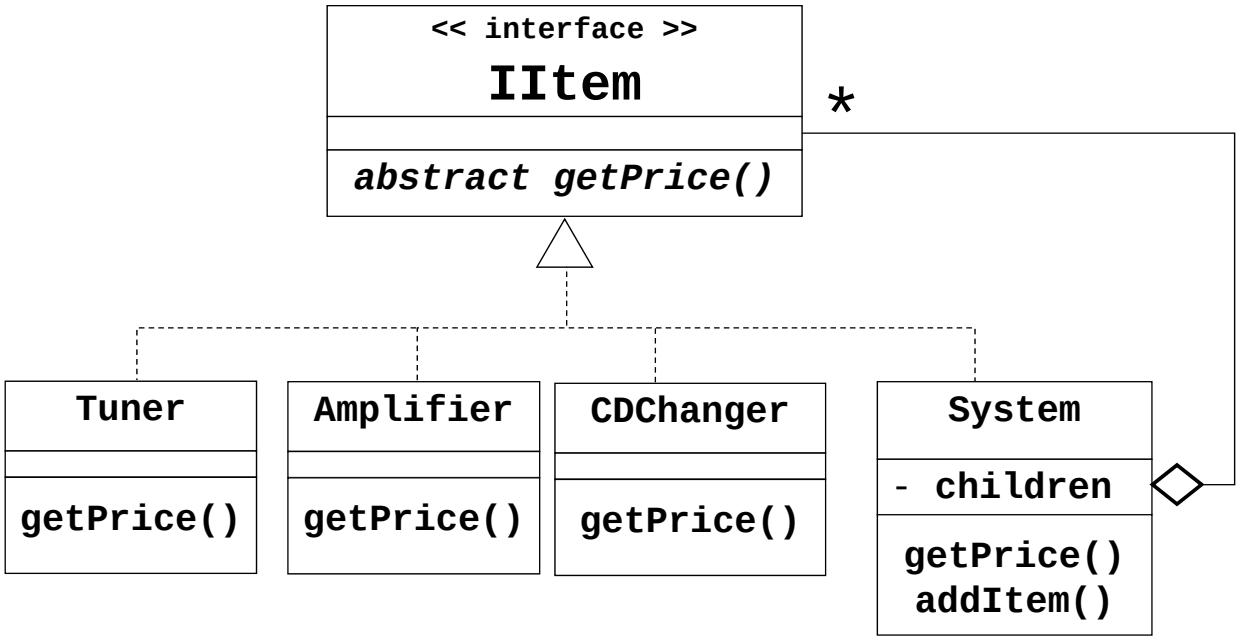
Solution



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**

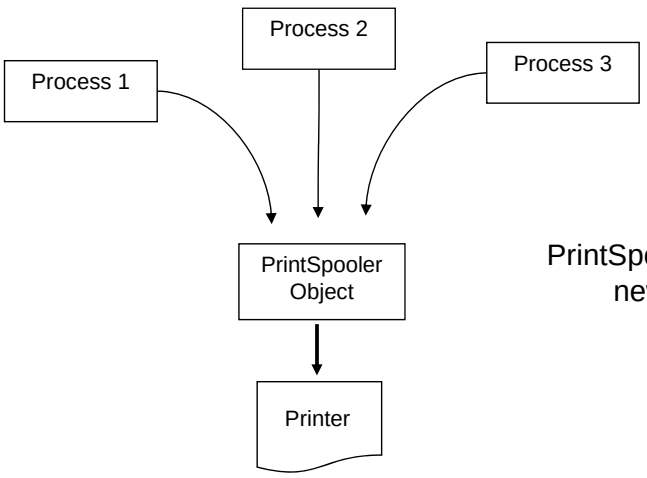
The Singleton Pattern

- **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();
```

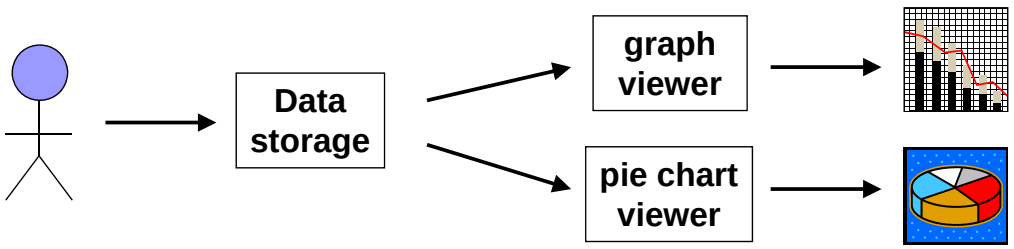
Singleton Implementation

```
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

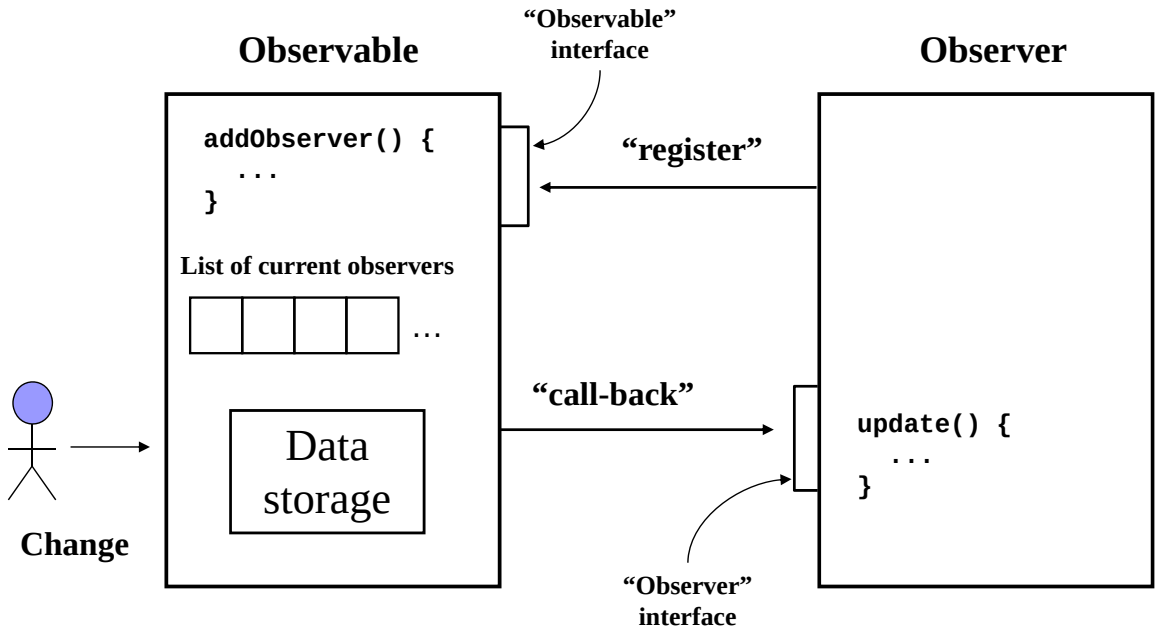
- An object stores data that changes regularly
- Various clients use the data in different ways
- Clients need to know when the data changes
- Code that is associated with the object that stores data should not need to change when new clients are added



"observable" object

"observers"

The `Observer` Pattern (cont.)



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 { //build-in CN1 interface
   public void update (Observable o, Object arg);
}
```

```
public interface IObservable { //user-defined interface
   public void addObserver (Observer obs);
   public void notifyObservers();
}
```

OR...

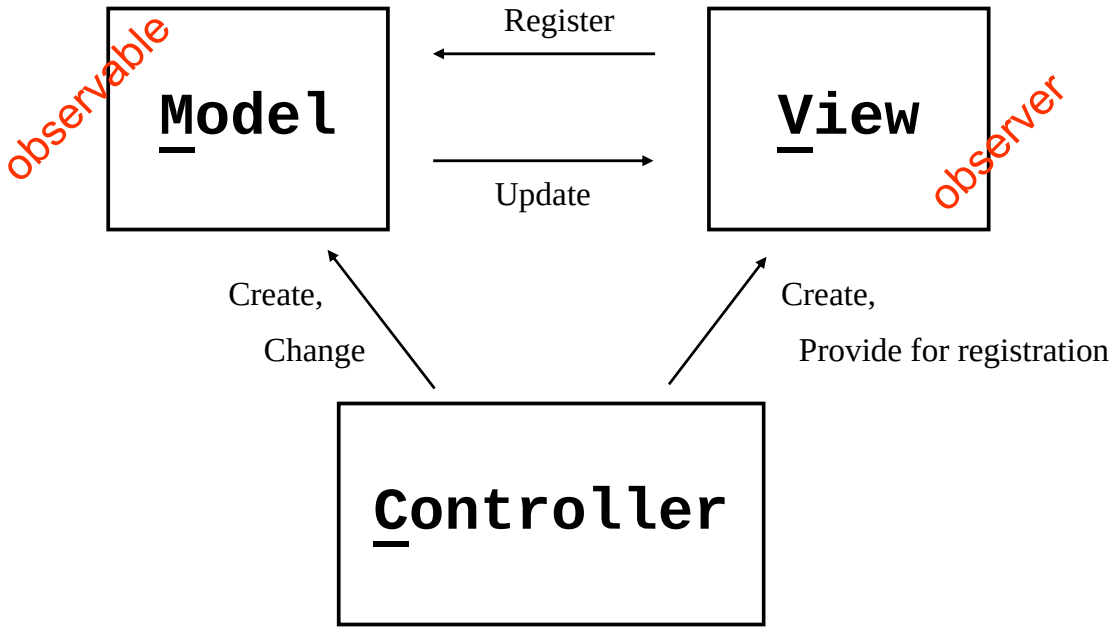
```
public class Observable extends Object { //build-in CN1 class
   public void addObserver (Observer obs) {...}
   public void notifyObservers() {...}
   protected void setChanged() {...}
   ...
}
```


Implementing Observer/Observable (cont.)

About extending from a build-in Observable class:

- Advantage: Provides code for **`notifyObservers()`** and **`addObserver()`**
- Disadvantage: You cannot extend from another class
- Make sure you call **`setChanged()`** before calling **`notifyObservers()`**
- **`notifyObservers()`** automatically calls **`update()`** on the list of observers that is created by **`addObserver()`**

MVC Architecture



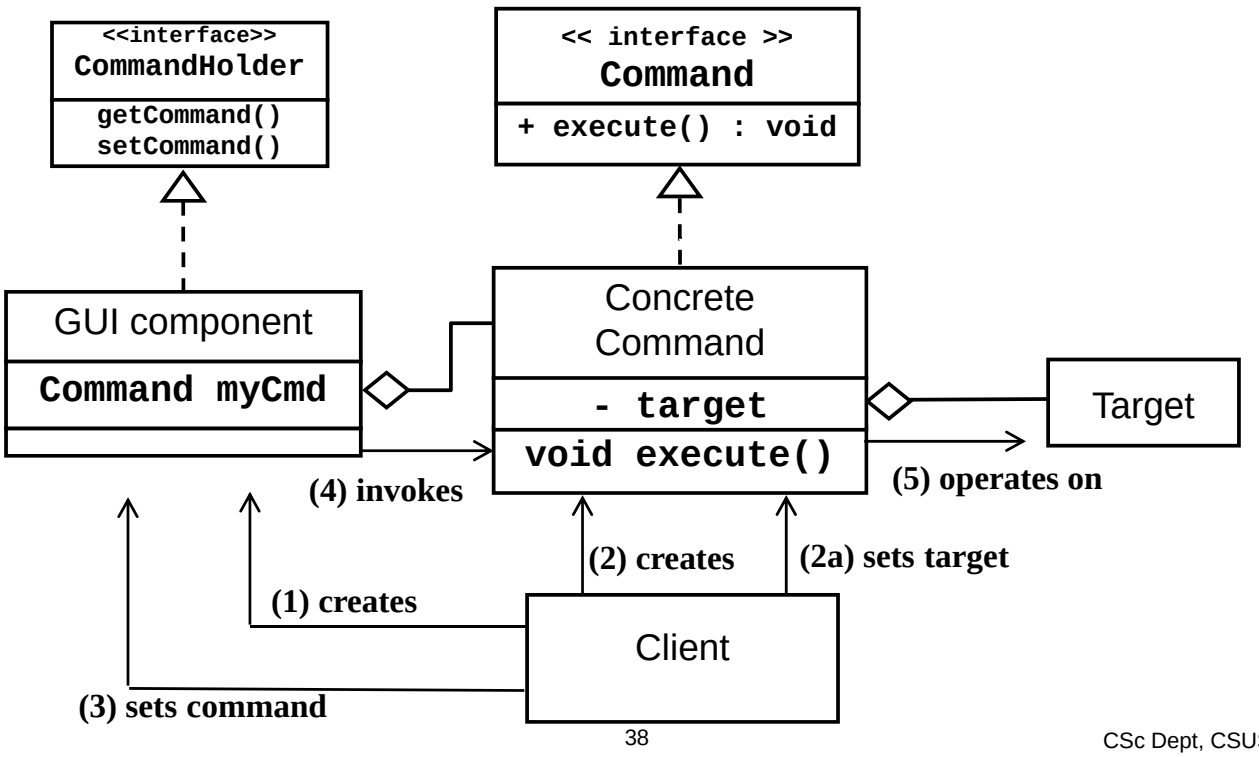

```
public class Controller {
  private Model model;
  private View v1;
  private View v2;
  public Controller () {
    model = new Model();    // create "Observable" model
    v1 = new View(model);   // create an "Observer" that registers itself
    v2 = new View();        // create another "Observer"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

public class Model extends Observable { // OR implements IObservable {
  // declarations here for all model data...
  // methods here to manipulate model data, etc.
  // if implementing IObservable, also provide methods that handle observer
  // registration and invoke observer callbacks
}

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 Command Pattern

Motivation

- Need to avoid having multiple copies of the code that performs the same operation invoked from different sources
- Desire to separate code implementing a command from the object which invokes it
- Need for maintaining *state information* about the command
  - Enabled or disabled?
  - Other data – e.g. invocation *count*

Command Pattern Organization



CN1 Command Class

- Implements **ActionListener** interface.
  - Provides empty body implementation for: **actionPerformed()** == "**execute()**"
  - We need to extend from **Command** and override **actionPerformed()** to perform the operation we would like to execute. In the constructor, do not forget to call super("command name")
- Also defines methods like: **isEnabled(), setEnabled(), getCommandName()**
- You can add a command object as a listener to a component using one of its **addXXXListener()** methods which takes **ActionListener** as a parameter (e.g. **addPointerPressedListener()** in **Component**, **addActionListener()** in **Button**, **addKeyListener()** in **Form**)
- When activated (button pushed, pointer/key pressed etc), component calls **actionPerformed()** method of its listener/command

CN1 Command Class (cont.)

Using the **addKeyListener()** of **Form**, we can attach a listener (an object of a listener class which implements **ActionListener** or an object of sub-class of **Command**) to a certain key.

This is called **key binding**: we are binding the listener/command (more specifically: the operation defined in its **actionPerformed()** method) to the key stroke, e.g:

```
/* Code for a form that uses key binding
//... [create a listener object called myCutCommand]
addKeyListener('c', myCutCommand);
//[when the 'c' key is hit, actionPerformed() method of CutCommand is called]
```

CN1 `Button` Class

`Button` is a "command holder"

- Defines methods like: **`setCommand(), getCommand()`**
- If you use **`setCommand()`** you do not need to also call **`addActionListener()`**since the command is *automatically added as listeners on the button*
- **`setCommand()`** changes the label of the button to the "command name" specified in command's construction

To use the command design pattern properly on buttons, add the command object to the button using **`setCommand()`** (instead of **`addActionListener()`**).

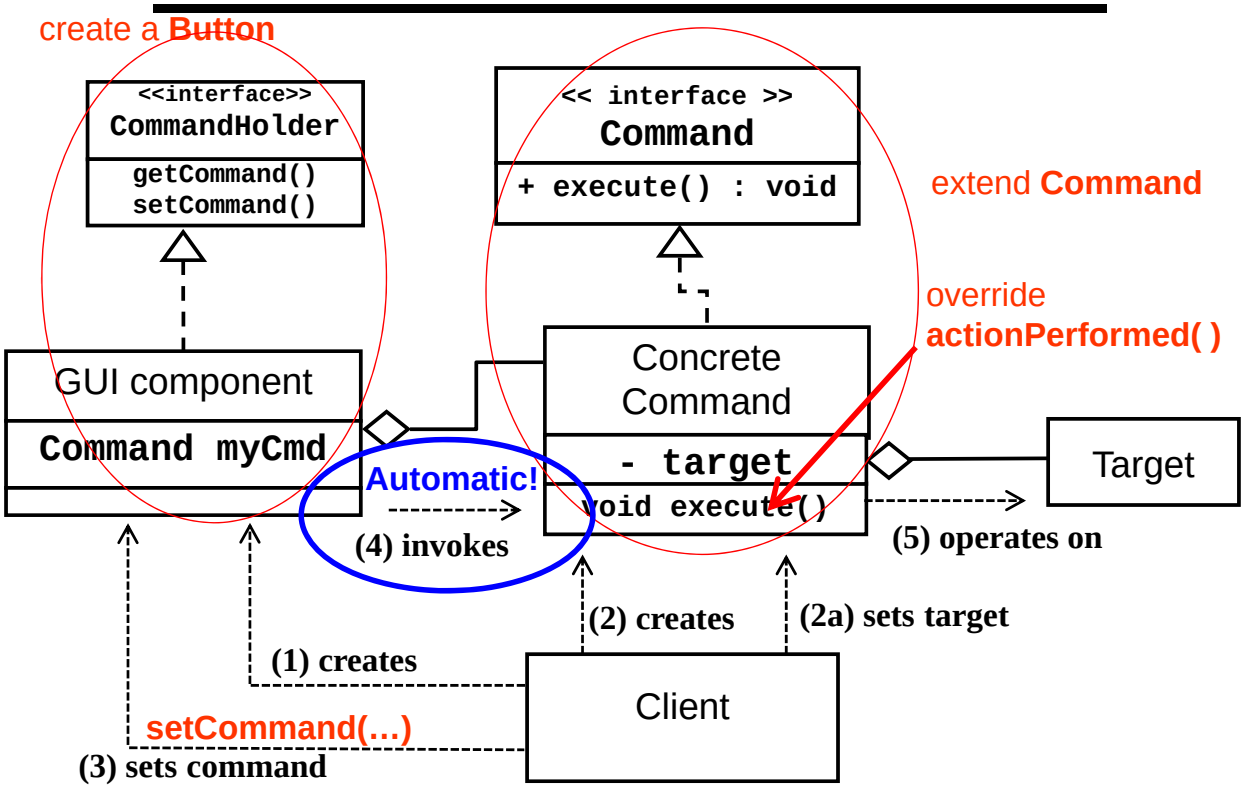
Remember **`CheckBox`** is-a **`Button`** too!

Adding Commands to Title Bar

When you add a regular command (e.g., without specifying a "SideComponent" property) to the title bar area using **`Toolbar`**'s **`addCommandToXXX()`** methods:

- an item (side/overflow menu item or a title bar area item) is automatically generated and added to the title bar area
- The command automatically becomes the listener of the item

Command Pattern – CN1



Summary of Implementing Command Design Pattern in CN1

- **Define your command classes:**
  - Extend **`Command`** (which implements **`ActionListener`** interface and provides empty body implementation of **`actionPerformed()`**)
  - Override **`actionPerformed()`**
- **Add a Toolbar and buttons to your form**
- **Instantiate command objects in your form**
- **Add command objects to various entities:**
  - buttons w/ **`setCommand(),`** title bar area items w/ **`Toolbar's addCommandToXXX()`** methods, key strokes w/ **`Form`**'s **`addKeyListener()`**

Implementing Command Design Pattern in CN1

```
/** This class instantiates several command objects, creates several GUI
 * components (button, side menu item, title bar item), and attaches the command objects
 * to the GUI components and keys.  The command objects then automatically get invoked
 * when the GUI component or the key is activated.
 */

public class CommandPatternForm extends Form {
  public CommandPatternForm () {
    //...[set a Toolbar to form]
   Button buttonOne = new Button("Button One");
   Button buttonTwo = new Button("Button Two");
    //...[style and add two bottons to the form]
   //create command objects and set them to buttons, notice that labels of buttons
   //are set to command names
   CutCommand myCutCommand = new CutCommand();
   DeleteCommand myDeleteCommand = new DeleteCommand();
   buttonOne.setCommand(myCutCommand);
   buttonTwo.setCommand(myDeleteCommand);
    //add cut commnand to the right side of title bar area
   myToolbar.addCommandToRightBar(myCutCommand);
   //add delete command to the side menu
   myToolbar.addCommandToSideMenu(myDeleteCommand);
    //bind 'c' ket to cut command and 'd' key to delete command
   addKeyListener('c', myCutCommand);
   addKeyListener('d', myDeleteCommand);
   show();
   }
}
```


Implementing Command Design Pattern in CN1 (cont.)

```
/** These classes define a Command which perform "cut" and “delete” operations.
 *  The commands are implemented as a subclass of Command, allowing it
 *  to be added to any object supporting attachment of Commands.
 *  This example does not show how the “Target” of the command is specified.
 */

public class CutCommand extends Command{
  public CutCommand() {
    super("Cut"); //do not forget to call parent constructor with command_name
  }
  @Override //do not forget @Override, makes sure you are overriding parent method
  //invoked to perform the 'cut' operation
  public void actionPerformed(ActionEvent ev){
    System.out.println("Cut command is invoked...");
  }
}
public class DeleteCommand extends Command{
  public DeleteCommand() {
    super("Delete");
  }
  @Override
  public void actionPerformed(ActionEvent e){
   System.out.println("Delete command is invoked...");
  }
}
```

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?

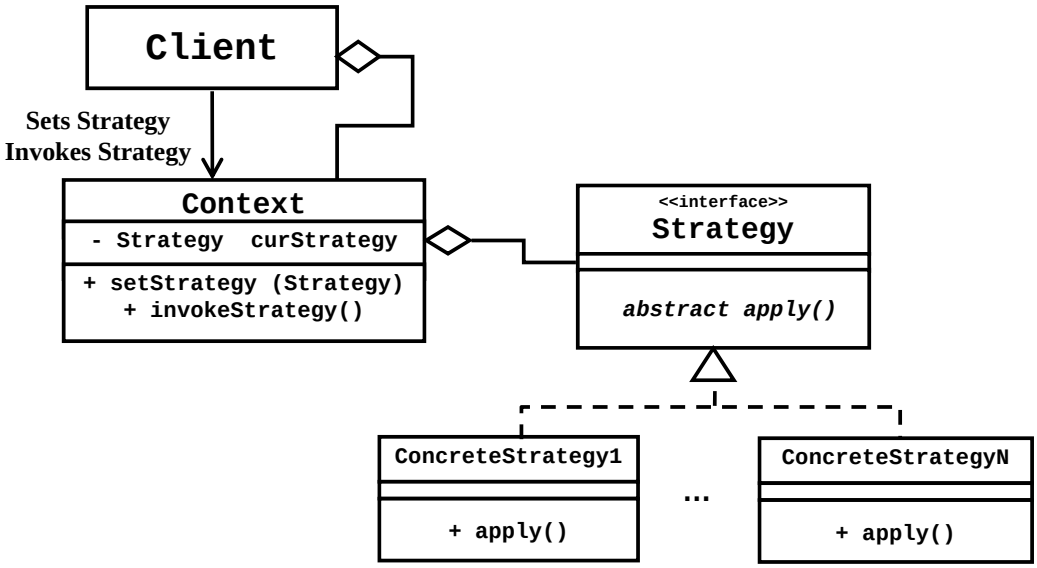
Changing or adding a plan requires changing the client!

Solution Approach

- Provide various objects that know how to "apply strategy" (e.g. apply fight, fireWeapon, or castMagicSpell strategies)
  - Each in a different way, but with a uniform interface
- The context (e.g. NPC) maintains a "current strategy" object
- Provide a mechanism for the client (e.g. Game) to *change* and *invoke* the current strategy object of a context

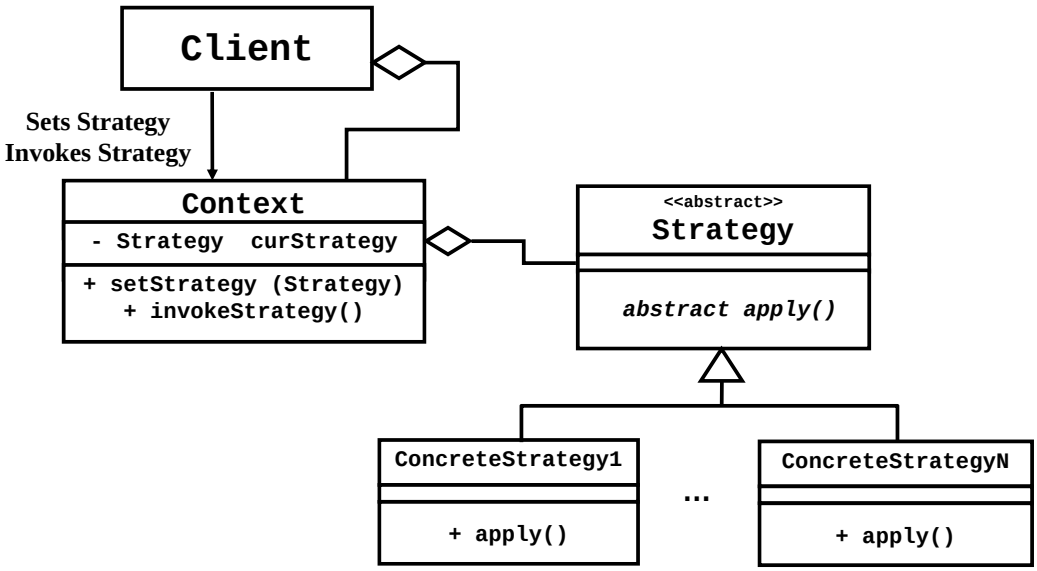
Strategy Pattern Organization

- **Using Interfaces**



Strategy Pattern Organization (cont.)

- **Using subclassing**



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

```
public class Warrior extends Character {

//code here for Warrior specific methods

}
```

```
public class Shaman extends Character {

//code here for Shaman specific methods

}
```

```
public class Hunter extends Character {
    private int bulletCount ;

    public boolean isOutOfAmmo() {
        if (bulletCount <= 0) return true;
        else return false;
    }
    public void fireWeapon() {
        bulletCount -- ;
    }

//code here for other Hunter specific
//methods

}
```

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());
                }
            }
        }
    }
}
```


CN1 Layouts

- Strategy abstract super class:

  `Layout`

- Client is the **`Form`**
- Context: **`Container`** (e.g., **`ContentPane`** of **`Form`**)
- Context methods:

  `public void setLayout (Layout lout)`
  `public void revalidate()`

- Concrete strategies (**`extends Layout`**):

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

- "Apply" method (declared in the **`Layout`** super class):

  abstract **`void layoutContainer(Container parent)`**

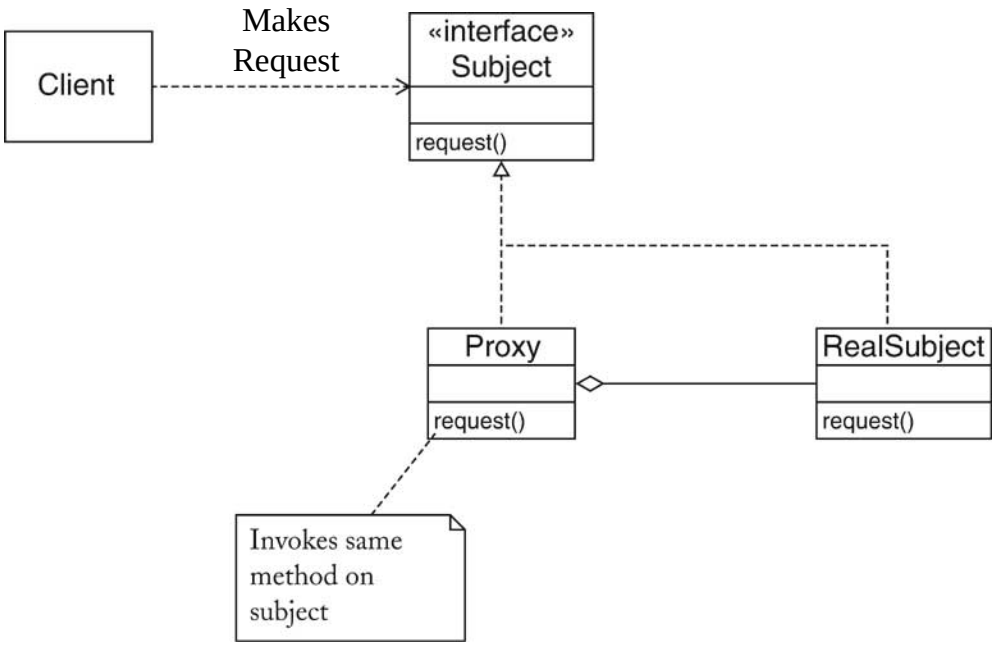
The Proxy Pattern

- 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 ; }
     //...[also has methods implementing IObservable]
}
```

Proxy Example (cont.)

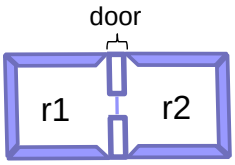
```
/** This class defines a Game containing a GameWorld with a ScoreView Observer. */
public class Game {
    public Game() {
        GameWorld gw = new GameWorld();   //construct a GameWorld
        ScoreView sv = new ScoreView();   //construct a ScoreView
        gw.addObserver(sv);               //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 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() ;}
    ...
}
```

8 - GUI Basics

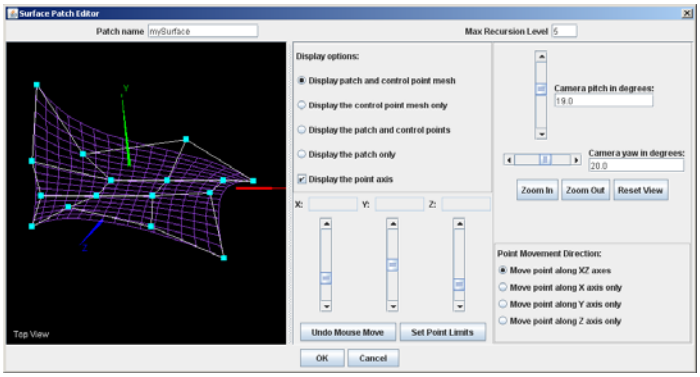
Computer Science Department
California State University, Sacramento

Overview

- **Displays and Color**
- **The UI Package of CN1**
- **UI Components: Form, Button, Label, Checkbox, ComboBox, TextField …**
- **Layout Managers**
- **Containers**
- **Side Menus**

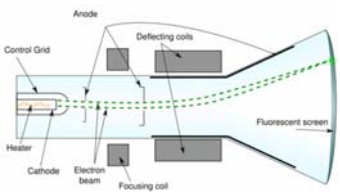
Modern Program Characteristics

- Graphical User Interfaces ("GUIs")
- "Event-driven" interaction



Common Display Types

- CRT (Cathode Ray Tube)



- LCD (Liquid Crystal Display)

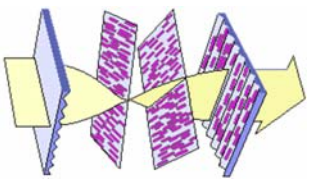
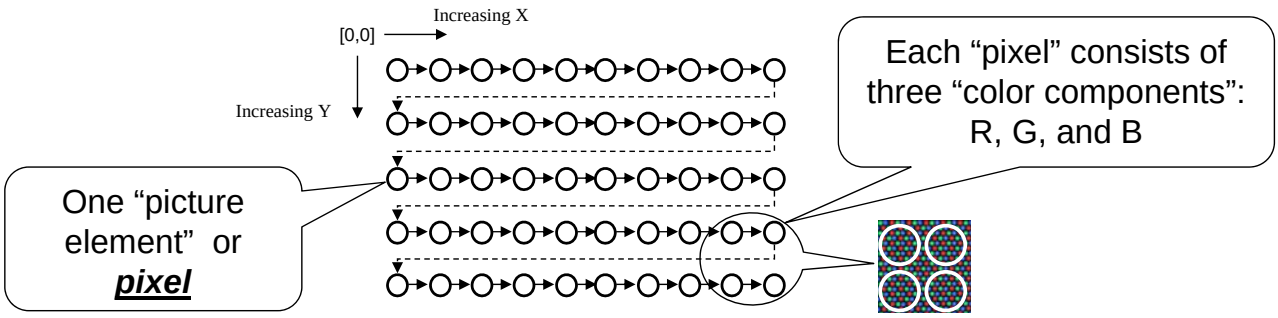


Image credits: Wikimedia Commons (http://commons.wikimedia.org); http://www.pctechguide.com/07panels.htm

Raster vs. Random Scan Devices

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



RGB Additive Color Model

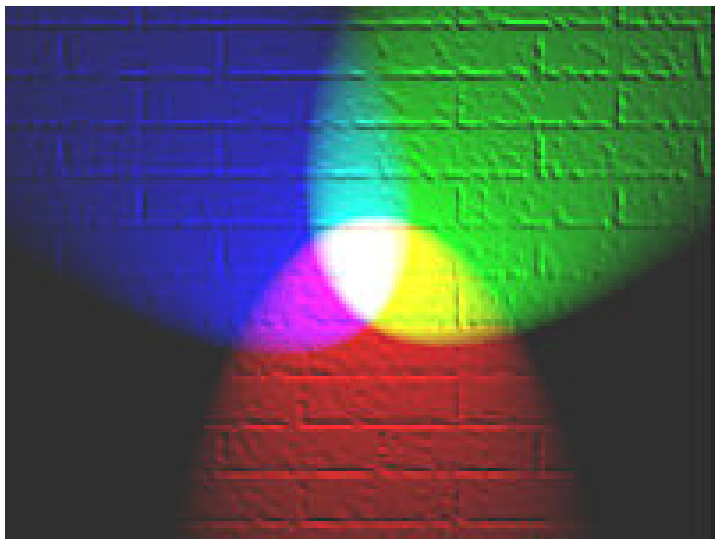


Image credit: http://en.wikipedia.org/wiki/RGB_color_model

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

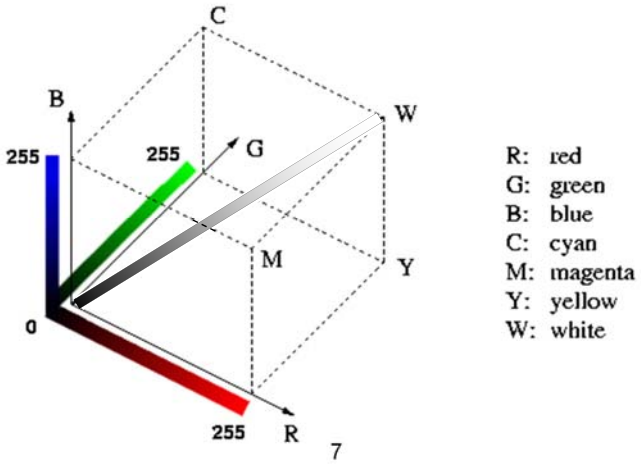
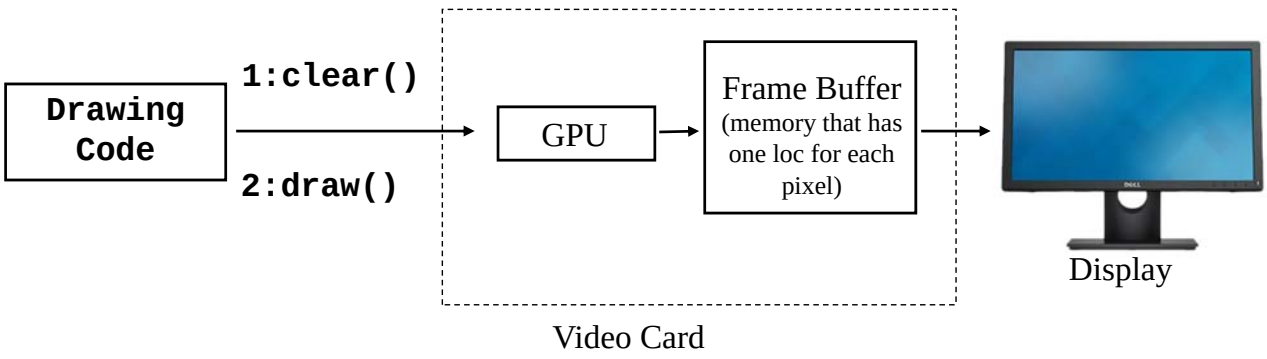


Image credit: http://gimp-savvy.com

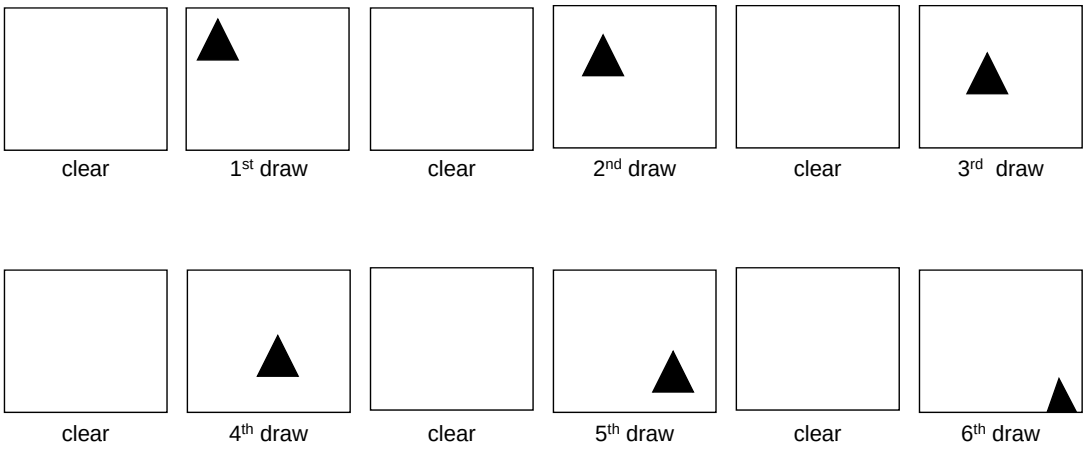
Frame Buffers

- Graphical Processing Unit (GPU) processes the commands sent from the drawing code and writes to the "*frame buffer*"
- Video card refreshes the screen from the frame buffer



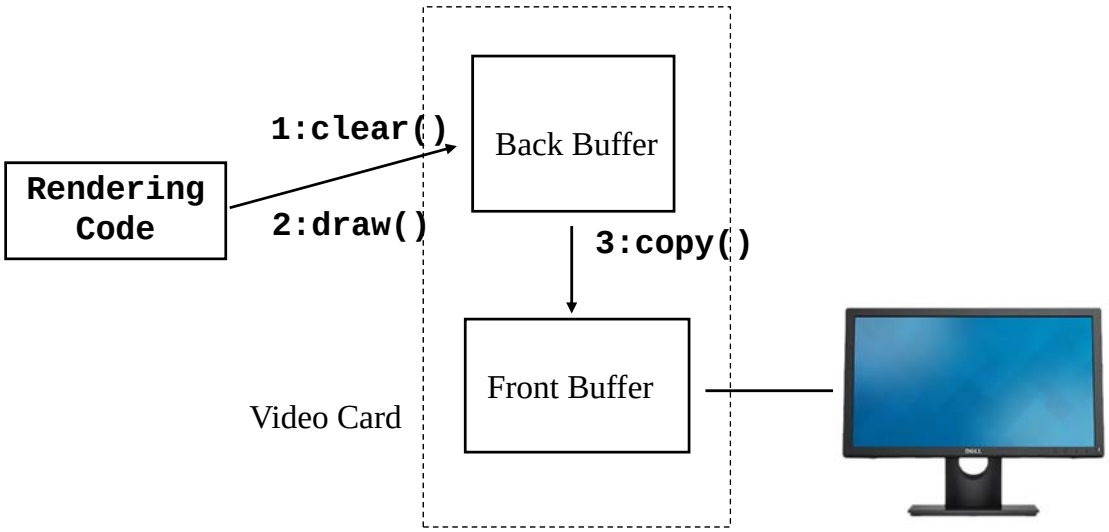
Flicker

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



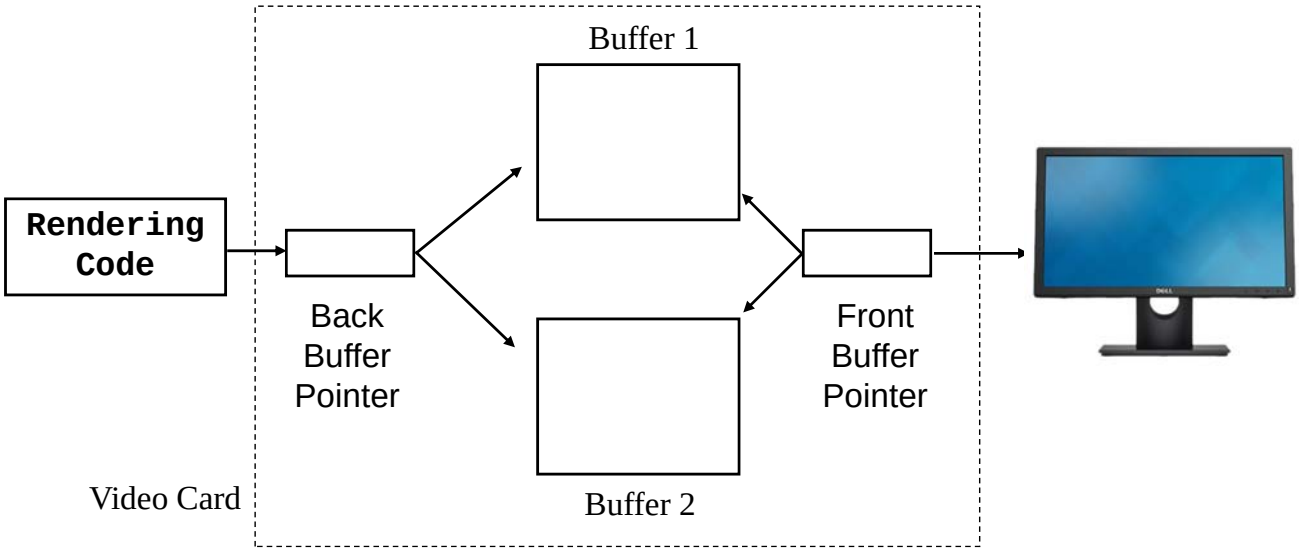
Double-Buffering

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



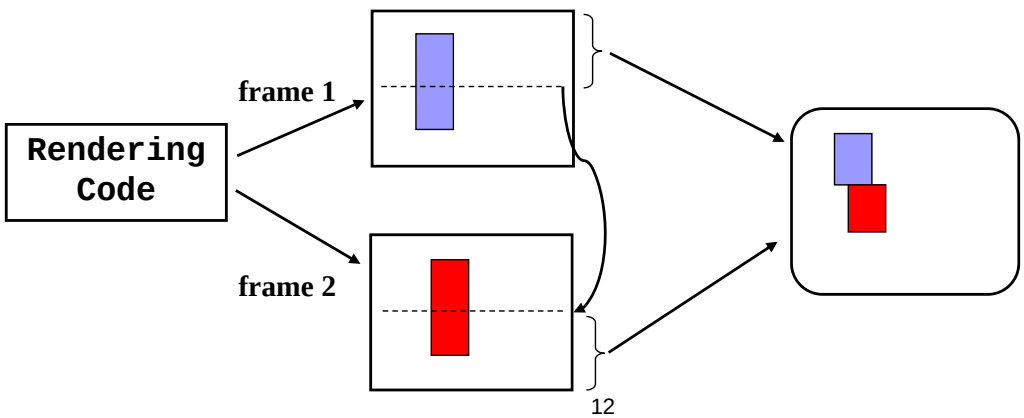
Page-Flipping

- Avoid copy() by changing a *pointer*



Tearing

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



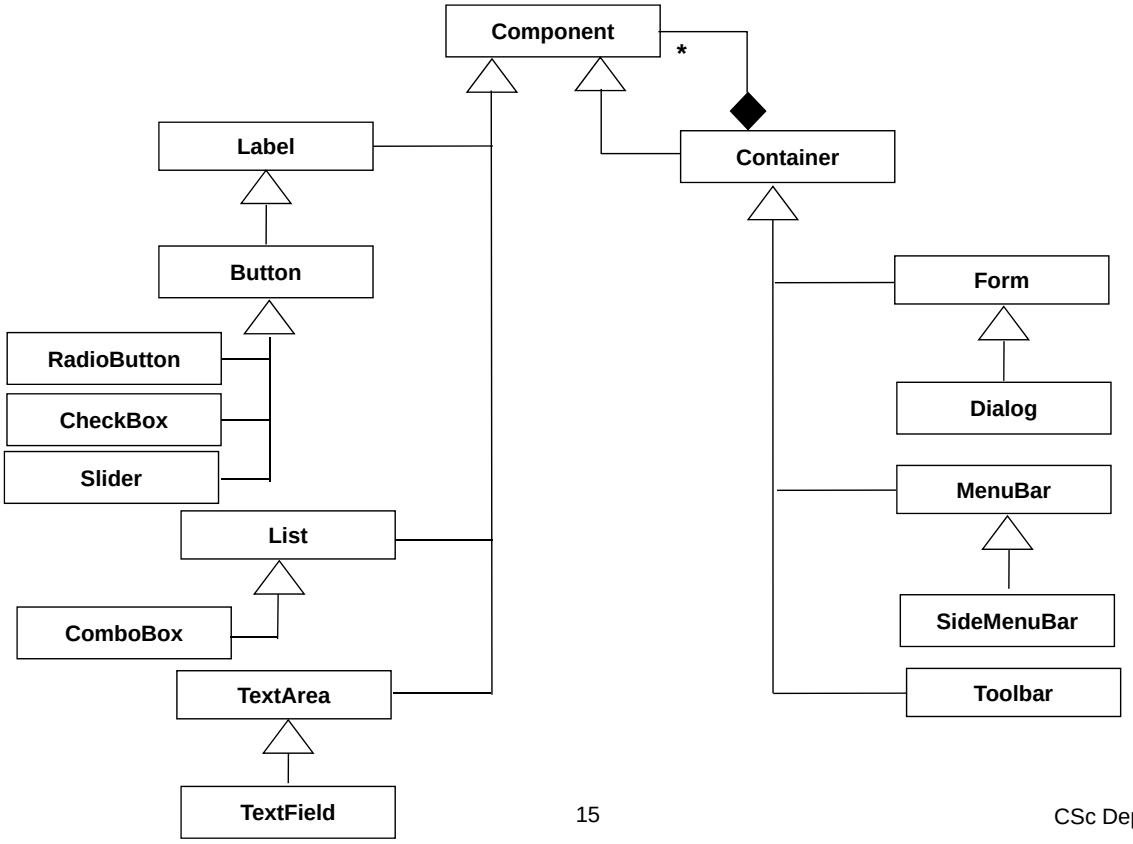
GUI Frameworks

- Collection of classes that take care of low-level details of drawing "things" on screen. Provides:
  - A *set of reusable screen components*
    - "Component": an object having a *graphical representation*
    - Usually has the ability to *interact* with the user
  - An *event mechanism* connecting "actions" to "code"
  - *Containers* and *Layout Managers* for arranging things on screen
  - Some other packages…

Examples of GUI Frameworks

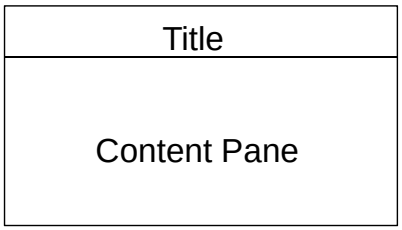
- Microsoft Foundation Classes (**MFC**): designed for C++ development on Windows (it is not build-in to C++)
- **AWT**: Java's first (inefficient) build-in GUI package
- JFC/**Swing**: Java's efficient build-in GUI package
- **UI**: CN1's GUI package (very similar to Swing)
- "Things" are called controls (MFC), components (AWT/Swing/CN1), widgets (X-Windows on Linux)

Important CN1- UI Components



Creating a Form in CN1

- The top-level container of CN1 (like **`JFrame`** in Swing)
- Only one form can be visible at any given time
- Form contains title and a content pane (and optionally a menu bar which we will not utilize in the assignments):



- Calling to **`myForm.addComponent()`** is actually invoking **`myForm.getContentPane().addComponent()`**
- Hence, content pane is the “parent” container of all components you add to the form.

Creating a Form in CN1 (cont.)

```
// Contents of File  DemoSimpleForm.java:
/** This class is a driver for running the SimpleForm class. It creates a Form.
It is the "Main" class of CN1 project (created with "native" theme and "Hello
World(Bare Bones)" template).
*/
//default import statements...
public class DemoSimpleForm {
private Form current;
//default implementations of methods like init(), stop(), destroy() ...
  public void start() {
    if(current != null){
              current.show();
              return;
            }
    //change the default implementation of start()
    new SimpleForm();
  }
}
```


Creating a Form in CN1 (cont.)

```
// Contents of File  SimpleForm.java:
import com.codename1.ui.Form;
/** This class creates a simple "Form"  by extending an existing
 *  class "Form", defined in the CN1's UI package.
 */
public class SimpleForm extends Form{
  public SimpleForm() {
    this.show();
  }
```

Titled Form in CN1

```
import com.codename1.ui.*;
/** This class creates a "Form" that has a title specified by the user
 *  User types the title on a “TextField” on a “Dialog”
 */
public class TitledForm extends Form {
    public TitledForm() {
       Command cOk = new Command("Ok");
       Command cCancel = new Command("Cancel");
       Command[] cmds = new Command[]{cOk, cCancel};
       TextField myTF = new TextField();
       Command c = Dialog.show("Enter the title:", myTF, cmds);
       //[if you only want to display the okay option, you do not need to
       //create “cmds”, just use Dialog.show("Enter the title:", myTF, cOk);]
       if (c == cOk)
         this.setTitle(myTF.getText());
       else
         this.setTitle("Title not specified");
       this.show();
    }
}
```


Closing App in CN1

```
import com.codename1.ui.*; //not listed in the rest of the examples
/** This class creates a "Form" that has a title “Closing App Demo”
 *  Then it pops up a “Dialog” confirming closing of the application
 */
public class ClosingApp extends Form {
  public ClosingApp() {
    this.setTitle("Closing App Demo");
    Boolean bOk = Dialog.show("Confirm quit", "Are you sure you want to quit?",
"Ok", "Cancel");
    //[in a dialog if you only want to display the okay option,
    //use Dialog.show("Title of dialog", "Text to display on dialog", "Ok", null);]
    if (bOk){
         //instead of System.exit(0), CN1 recommends using:
         Display.getInstance().exitApplication();
         }
    this.show();
  }
}
```

CN1 `Display` class

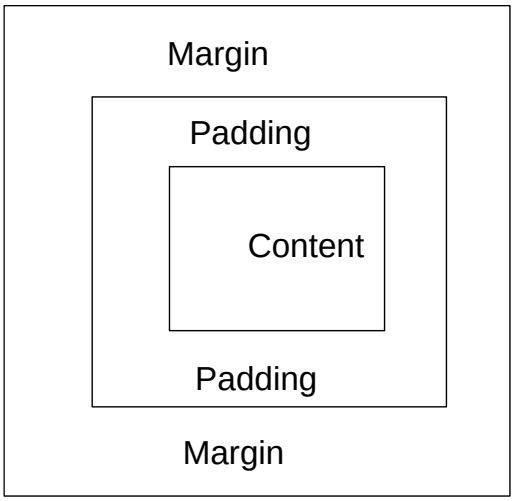
- Central class that manages rendering/events and is used to place top level components (Form) on the display.
- Has static **`getInstance()`** method which return the **`Display`** instance.
- To get the resolution of your display, you can call: **`Display.getInstance().getDisplayWidth()`** or …**`Height()`**
- **`Display.getInstance().getCurrent()`** return the form currently displayed on the screen or null if no form is currently displayed.

Adding Components to Form

```
public class FormWithComponents extends Form {
  public FormWithComponents () {
  // create a new label object
  Label myLabel = new Label("I am a Label");
  // add the label to the "content pane" of the form
  this.getContentPane().addComponent(myLabel);
  // [you can also call this.addComponent(myLabel) or simply this.add(myLabel)]
  // create a button and add
  Button myButton = new Button("I am a Button");
  this.addComponent(myButton);
  // create a checkbox and add
  CheckBox myCheck = new CheckBox("I am a CheckBox");
  this.addComponent(myCheck);
  // add a combo box (drop-down list) and add
  ComboBox myCombo = new ComboBox("Choice 1","Choice 2","Choice 3");
  this.addComponent(myCombo);
  this.show();
  }
}
```

CN1 `Style` class

Represents the look of a given component: colors, fonts, transparency, margin and padding & images.



Setting style of a Component

```
public class ComponentsWithStyle extends Form {
  public ComponentsWithStyle () {
    Button button1 = new Button("Plain button");
    Button button2 = new Button("Button with style");
    //change background and foreground colors of the unselected style of the button
    button2.getUnselectedStyle().setBgTransparency(255);
    button2.getUnselectedStyle().setBgColor(ColorUtil.BLUE);
    button2.getUnselectedStyle().setFgColor(ColorUtil.WHITE);
    //[use button2.getAllStyles() to set all styles (selected, pressed, disabled, etc.) of
the component at once]
    //add padding to all styles of button2
    button2.getAllStyles().setPadding(Component.TOP, 10);
    button2.getAllStyles().setPadding(Component.BOTTOM, 10);
    //[you can also add padding to left and right by using Component.LEFT and
Component.RIGHT]
    addComponent(button1);
    addComponent(button2);
    show(); //not listed in the rest of the examples
  }
}
```

Setting style of a Component (cont.)

```
public class ComponentsWithStyle extends Form {
  public ComponentsWithStyle () throws IOException { //for Image.createImage()
   //add button1 and button2 as shown in the previous example
   //set a background image for all styles of the form
   InputStream is = Display.getInstance().getResourceAsStream(getClass(),
                                                        "/BGImage.jpg");
   Image i = Image.createImage(is);
   this.getAllStyles().setBgImage(i);
   //set an image for the unselected style of the button
   Button button3 = new Button("Expand");
   button3.getAllStyles().setPadding(Component.TOP, 10);
   //[if necessary, also add padding to bottom, left, right, etc]
   is = Display.getInstance().getResourceAsStream(getClass(), "/expand.gif");
   //[copy the images directly under "src" directory]
   i = Image.createImage(is);
   button3.getUnselectedStyle().setBgImage(i);
   addComponent(button3);
   }
}
```


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:

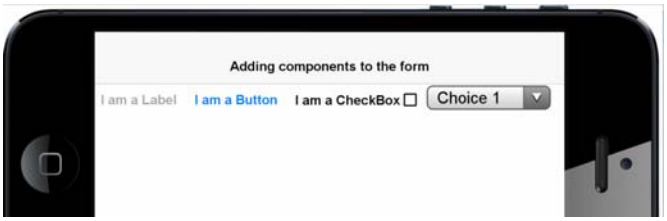
```
myContainer.setLayout( new BorderLayout() );
```

- Components can have a preferred *size*
  - **`setPreferredSize()`** of **`Component`** is depreciated
  - override **`calcPeferredSize()`** of **`Component`** to reach similar functionality (do not use this in the assignments)
  - Layout managers *may or may not* respect preferred size either entirely or partially (e.g., **`FlowLayout`** respects it whereas **`BoxLayout`** does not respect it entirely…)

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*
  - You can center components in the component by using:

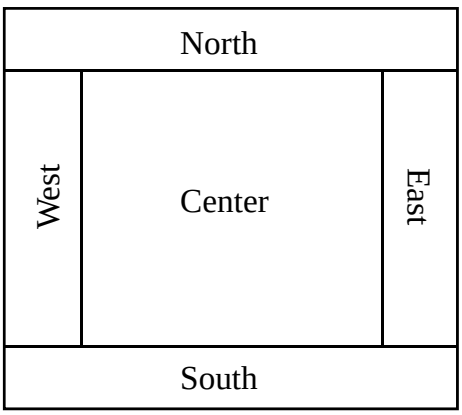
```
myContainer.setLayout(new FlowLayout(Component.CENTER));
```



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(BorderLayout.CENTER, myComponent);
```



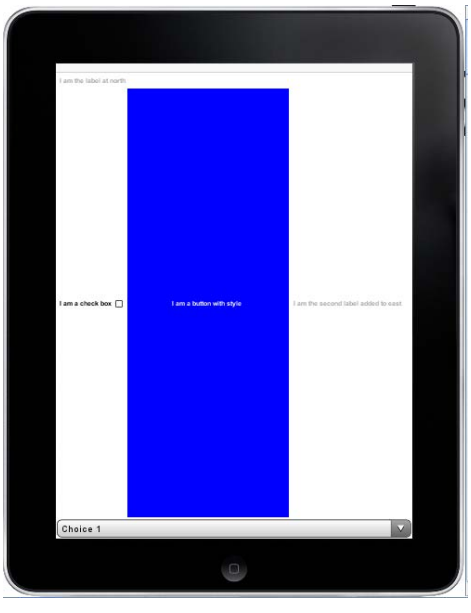
Layout Managers (cont.)

- **BorderLayout** (cont.)

```
public class BorderLayoutForm extends Form{//not listed in the rest
  public BorderLayoutForm() {            //of the examples
    //default layout for container is FlowLayout, change it to BorderLayout
     this.setLayout(new BorderLayout());
    //add a label to the top area of border layout
     Label myLabel = new Label("I am the label at north");
     this.add(BorderLayout.NORTH, myLabel);
    //... [add a check box to BorderLayout.WEST, a combo box to BorderLayout.SOUTH]
    //create a button to add to the center area
    Button myButton = new Button("I am a button with style");
     //...[set style of the button and add it to BorderLayout.CENTER]
    //add other labels to the left area of border layout
     Label myLabel2 = new Label("I am the first label added to east");
     this.add(BorderLayout.EAST, myLabel2);
    //[THIS LABEL WILL NOT BE VISIBLE, see upcoming slides for a solution]
     Label myLabel3 = new Label("I am the second label added to east");
     this.add(BorderLayout.EAST, myLabel3);}
}
```


Layout Managers (cont.)

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



Layout Managers (cont.)

- Example: **BoxLayout**
  - Adds components to a horizontal or a vertical line that doesn't break the line
  - Box layout accepts an axis in its constructor:

```
myContainer.setLayout(new BoxLayout(BoxLayout.X_AXIS));
myContainer.setLayout(new BoxLayout(BoxLayout.Y_AXIS));
```

  - Components are stretched along the opposite axis, e.g. X_AXIS box layout will place components horizontally and stretch them vertically.

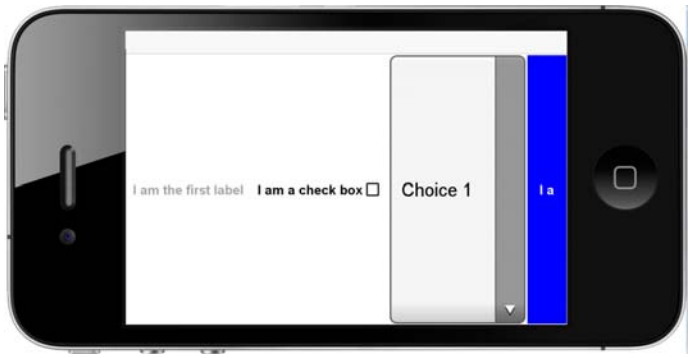
Layout Managers (cont.)

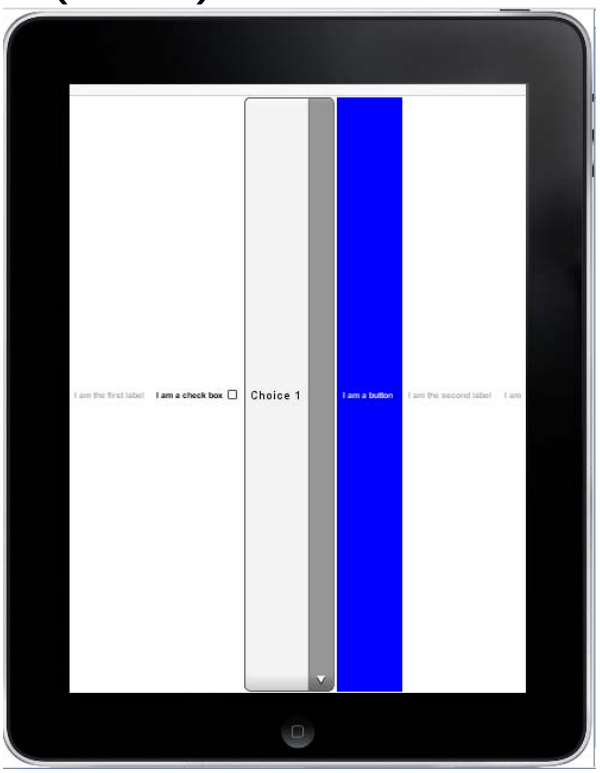
- Example: **BoxLayout** (cont.)

```
/* Code for a form with box layout */
setLayout(new BoxLayout(BoxLayout.X_AXIS));
//add a label as the first item
Label myLabel = new Label("I am the first label");
add(myLabel);
//... [add a check box as the second, a combo box as the third item
Button myButton = new Button("I am a button");
//...[set style of the button and add it as the fourth item]
//add other labels as fifth and sixth items
Label myLabel2 = new Label("I am the second label");
add(myLabel2);
Label myLabel3 = new Label("I am the third label");
add(myLabel3);
```

Layout Managers (cont.)

- Example: **`BoxLayout`** (cont.)





Layout Managers (cont.)

Setting preferred size (do not use this in the assignments, instead use **`setPadding()`** of **`Style`** class to change size of your buttons etc):

```
public class MyComponent extends Component{
@Override
protected Dimension calcPreferredSize(){
  return new Dimension(500, 300);}
public MyComponent() {
  //this is an empty component with a blue border
  this.getAllStyles().setBorder(Border.createLineBorder(2, ColorUtil.BLUE));}
}
```

`----------` below is the code for a form with default layout

```
//using default flow layout, first add a MyComponent
MyComponent myComponent = new MyComponent();
add(myComponent);
//then add several buttons with styles
```

`----------` below is the code for a form with box layout

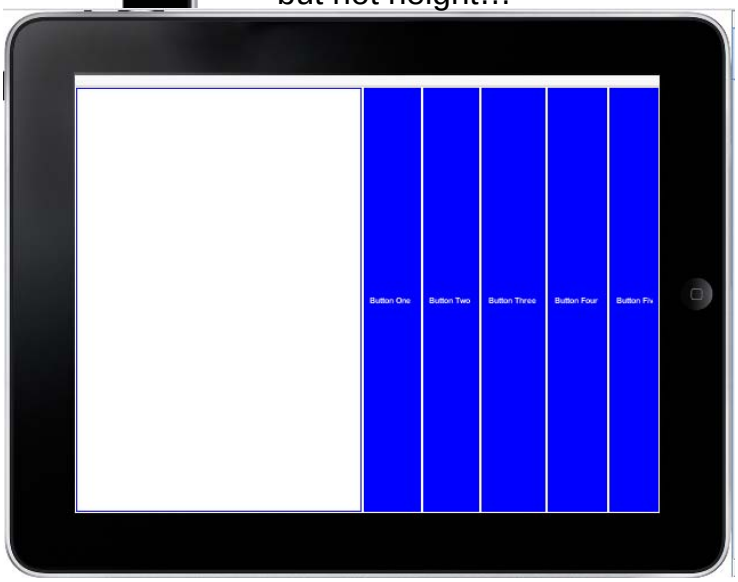
```
//using X_AXIS box layout
setLayout(new BoxLayout(BoxLayout.X_AXIS));
//add MyComponent to the first item, and then then add several buttons with styles
```

Layout Managers (cont.)

Setting preferred size (cont.):

FlowLayout (above) respects both preferred width and height…

BoxLayout (below) Respects preferred width but not height…



Layout Managers (cont.)

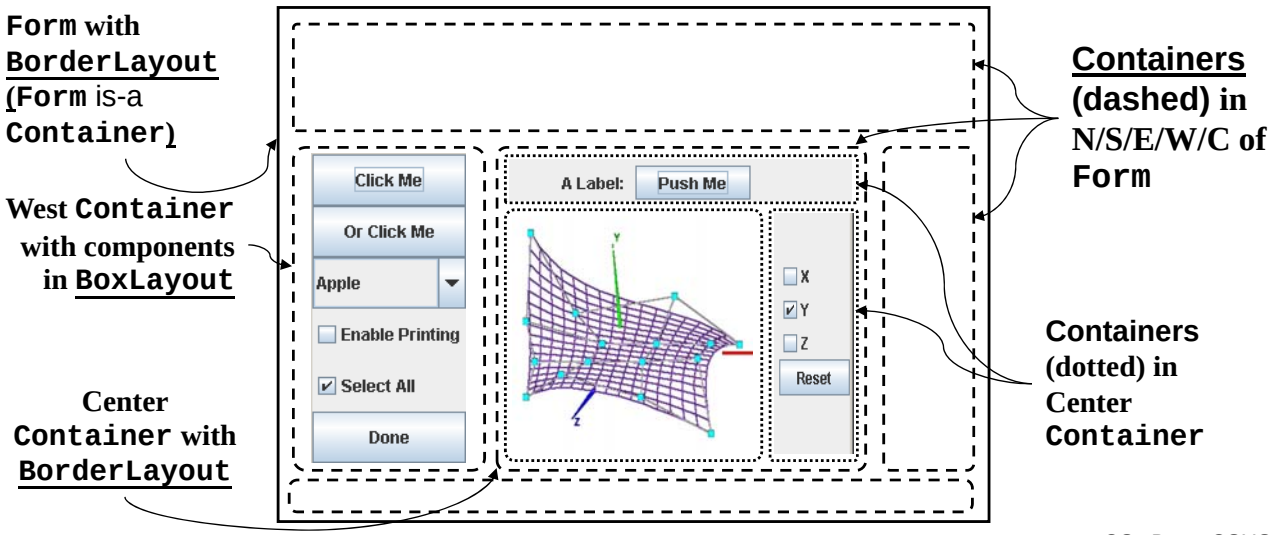
- Other Layout Managers
  - **`GridLayout`**
  - **`TableLayout`**
  - Etc..
- You can change the layout manager of the container in runtime:
  - Example of the *Strategy* Design Pattern

GUI Layout

GUIs usually have multiple “areas”

CN1 Container Class

- **`Container`** (like **`JPanel`** in Swing): an *invisible* component that…
  - Can be assigned to an area
  - Can have a layout manager assigned to it
  - Can hold other components (**`Container`** is-a **`Component`** and has-a **`Component`**)



Container Example

```
/* Code for a form with containers in different layout arrangements */
setLayout(new BorderLayout());
//top Container with the GridLayout positioned on the north
Container topContainer = new Container(new GridLayout(1,2));
topContainer.add(new Label("Read this (t)"));
topContainer.add(new Button("Press Me (t)"));
//Setting the Border Color
topContainer.getAllStyles().setBorder(Border.createLineBorder(4,
                                                ColorUtil.YELLOW));
add(BorderLayout.NORTH,topContainer);
//left Container with the BoxLayout positioned on the west
Container leftContainer = new Container(new BoxLayout(BoxLayout.Y_AXIS));
//start adding components at a location 50 pixels below the upper border of the container
leftContainer.getAllStyles().setPadding(Component.TOP, 50);
leftContainer.add(new Label("Text (l)"));
leftContainer.add(new Button("Click Me (l)"));
leftContainer.add(new ComboBox("Choice 1","Choice 2","Choice 3"));
leftContainer.add(new CheckBox("Enable Printing (l)"));
leftContainer.getAllStyles().setBorder(Border.createLineBorder(4,
                                                ColorUtil.BLUE));
add(BorderLayout.WEST,leftContainer);
    ... continued
```

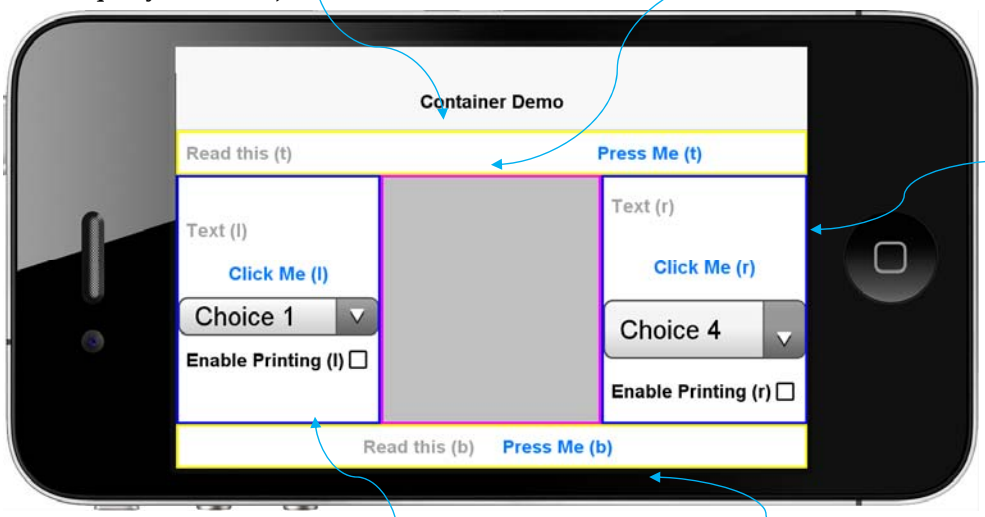

Container Example (cont.)

```
... continued
//right Container with the GridLayout positioned on the east
Container rightContainer = new Container(new GridLayout(4,1));
//...[add similar components that exists on the left container]
add(BorderLayout.EAST,rightContainer);
//add empty container to the center
Container centerContainer = new Container();
//setting the back ground color of center container to light gray
centerContainer.getAllStyles().setBgTransparency(255);
centerContainer.getAllStyles().setBgColor(ColorUtil.LTGRAY);
//setting the border Color
centerContainer.getAllStyles().setBorder(Border.createLineBorder(4,
                                                ColorUtil.MAGENTA));
add(BorderLayout.CENTER,centerContainer);
//bottom Container with the FlowLayout positioned on the south, components are laid out
//at the center
Container bottomContainer = new Container(new FlowLayout(Component.CENTER));
//...[add similar components that exists on the top container]
add(BorderLayout.SOUTH,bottomContainer);
```

Container Example – Output

`Container` in North with `GridLayout` (space is divided to two equally-sized cells)

Empty `Container` in Center with (with light gray background)



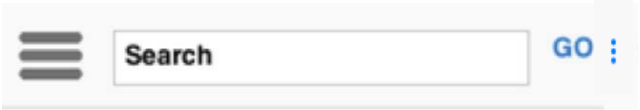
`Container` in East with `GridLayout` (space is divided to four equally-sized cells)

`Container (with padding)` in West with `BoxLayout` (components are positioned 50 pixels below the top border of the container)

`Container (with padding)` in South with `FlowLayout` (components are positioned at the center)

CN1 `Toolbar` class

- Provides deep customization of the title bar area of your form.

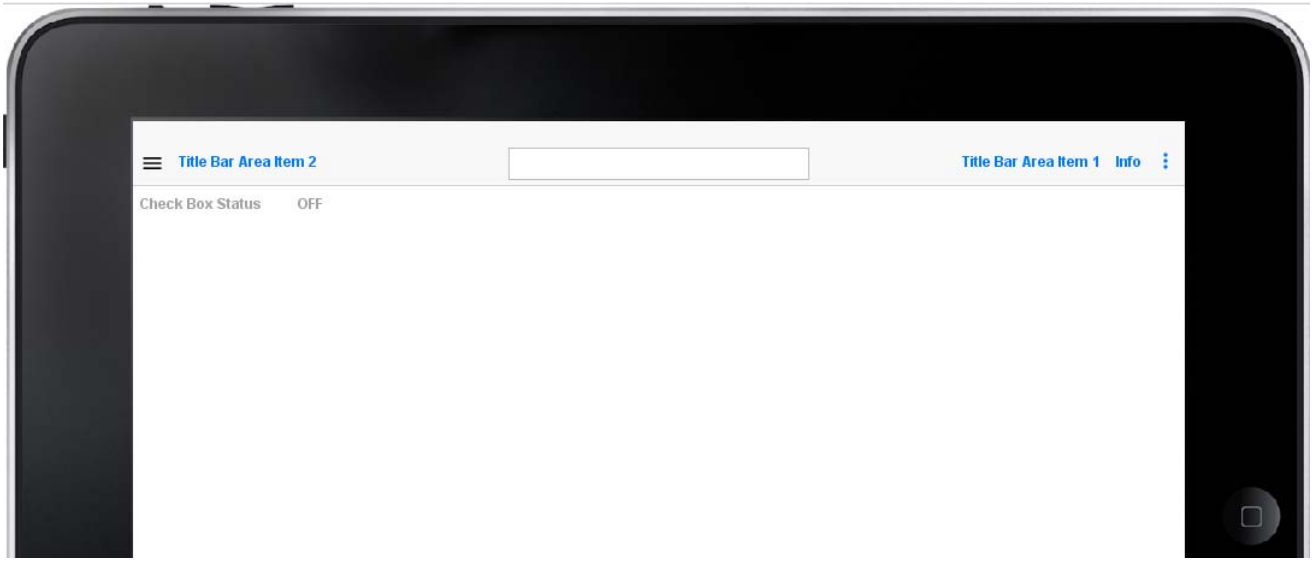


- Set it to your from with: **`myForm.setToolbar(toolbar)`**
- Allows adding commands to four locations:
  - **`addCommandToSideMenu()`** (to side menu: ≡)
  - **`addCommandToOverflowMenu()` `(to`** Android style menu: ⋮)
  - **`addCommandToRightBar()`** (to right of the title bar area)
  - **`addCommandToLeftBar()`** (to left of the title bar area)

Adding Items to Title Bar

```
/* Code for a form with a toolbar */
Toolbar myToolbar = new Toolbar();
setToolbar(myToolbar);//make sure to use lower-case "b", setToolBar() is depreciated
//add a text field to the title
TextField myTF = new TextField();
myToolbar.setTitleComponent(myTF);
//[or you can simply have a text in the title: this.setTitle("Adding Items to Title Bar");]
//add an "empty" item (which does not perform any operation) to side menu
Command sideMenuItem1 = new Command("Side Menu Item 1");
myToolbar.addCommandToSideMenu(sideMenuItem1);
//add an "empty" item to overflow menu
Command overflowMenuItem1 = new Command("Overflow Menu Item 1");
myToolbar.addCommandToOverflowMenu(overflowMenuItem1);
//add an "empty" item to right side of title bar area
Command titleBarAreaItem1 = new Command("Title Bar Area Item 1");
myToolbar.addCommandToRightBar(titleBarAreaItem1);
//add an "empty" item to left side of title bar area
Command titleBarAreaItem2 = new Command("Title Bar Area Item 2");
myToolbar.addCommandToLeftBar(titleBarAreaItem2);
//...[add other side menu, overflow menu, and/or title bar area items]
```


Adding Items to Title Bar (cont.)



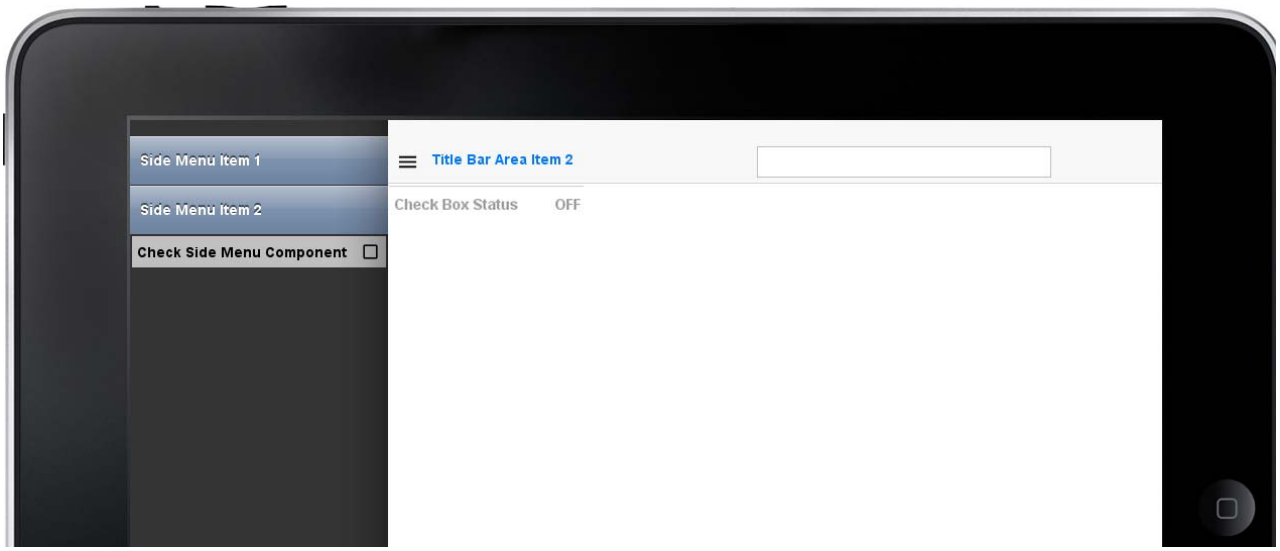
Complex Menus

- Menu items can contain components (like the title area):

```
/* Code for a form which has a CheckBox as a side menu item*/
//add a check box to side menu (which does not perform any operation yet..)
Command sideMenuItemCheck = new Command("Side Menu Item Check ");
CheckBox checkSideMenuComp = new CheckBox("Check Side Menu Component");
//set the style of the check box
checkSideMenuComp.getAllStyles().setBgTransparency(255);
checkSideMenuComp.getAllStyles().setBgColor(ColorUtil.LTGRAY);
//set "SideComponent" property of the command object to the check box
sideMenuItemCheck.putClientProperty("SideComponent", checkSideMenuComp);
//add the command to the side menu, this places its side component (check box) in the side menu
myToolbar.addCommandToSideMenu(sideMenuItemCheck);
```

- We will later see how to attach operations to commands and link them to the components in menus…

Complex Menus (cont.)



9 - Event-Driven Programming

Computer Science Department
California State University, Sacramento

Overview

- **Traditional vs. Event-Driven Programs**
- **Events**
- **Event Listeners:**
  - **CN1 `ActionListener` interface**
  - **Adding action/key/pointer listeners to components**
  - **Command design pattern, CN1 `Command` class, key bindings**
  - **Pointer handling**

Traditional vs. Event-Driven

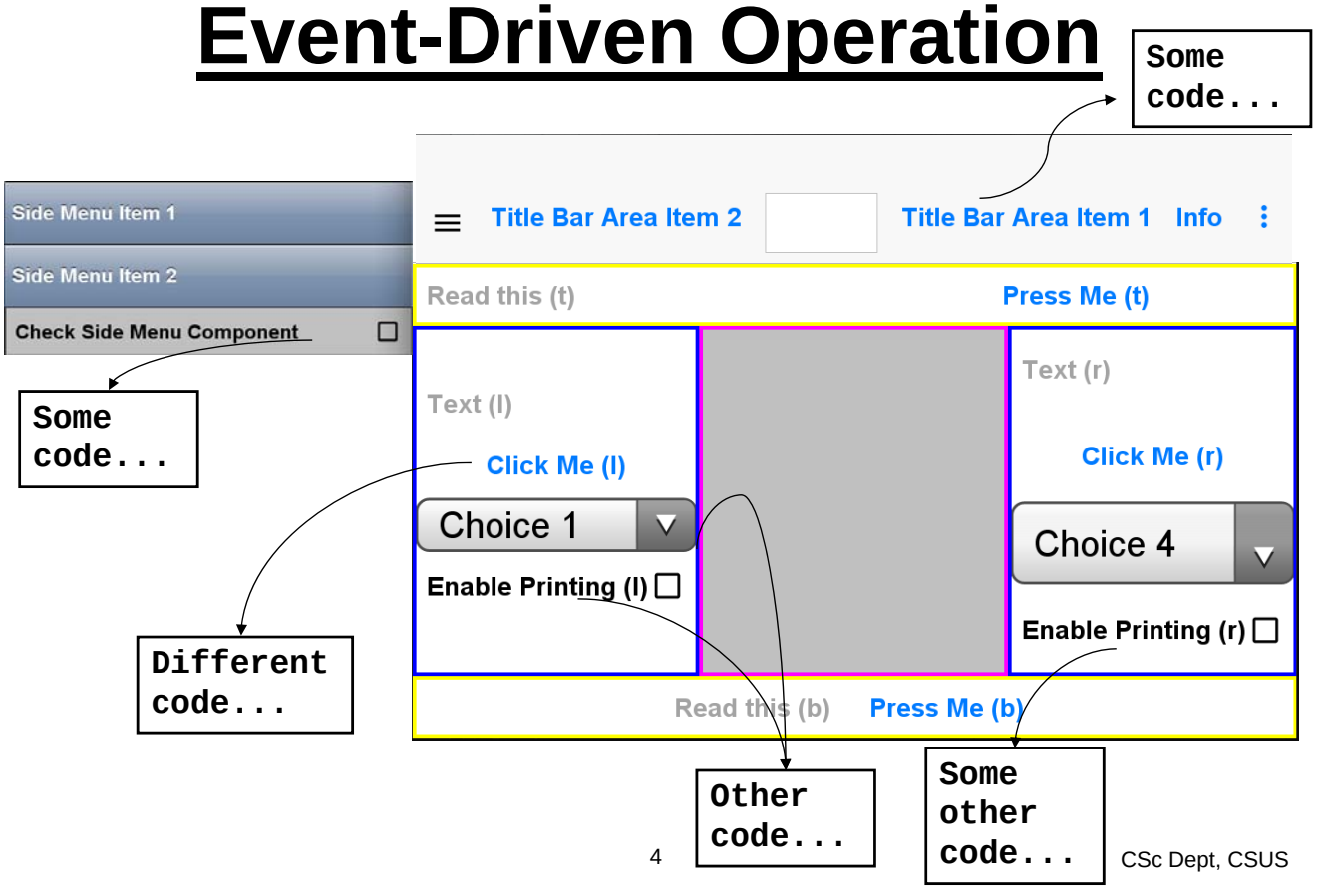
- Traditional program organization:

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

- Event-driven program organization:

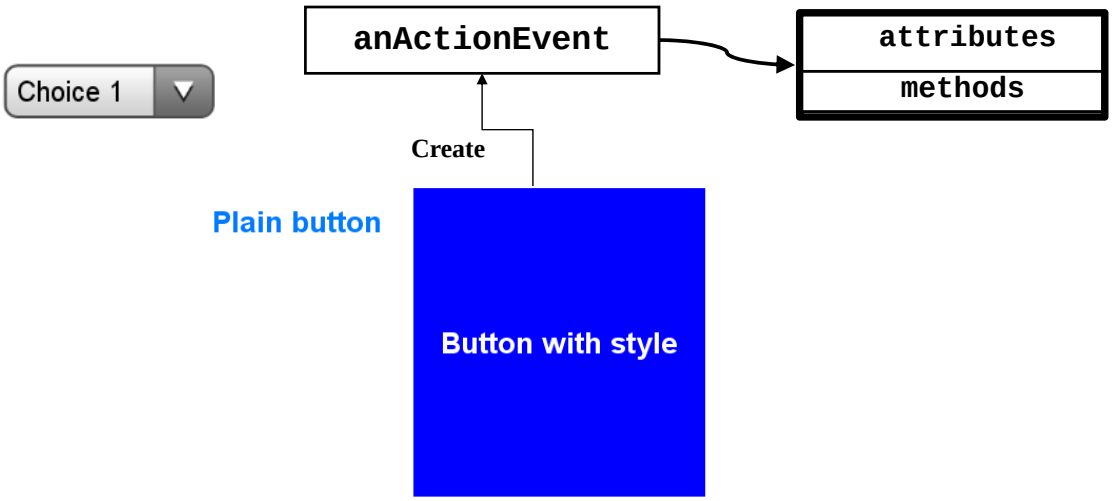
```
create a form ;
create some controls (buttons, etc.) ;
add controls to form ;
make the form visible ;
```


Event-Driven Operation



Event Objects

Activating a component and use of keys and the pointer create an object of type **`ActionEvent`**



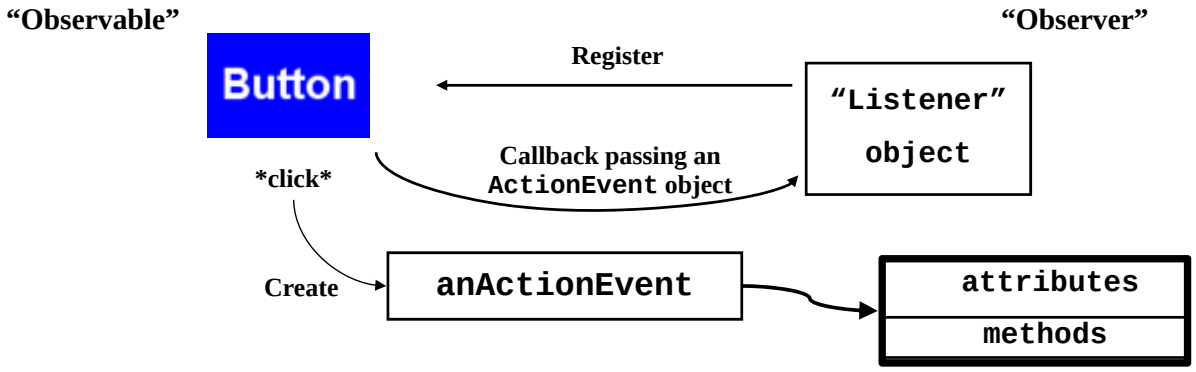
Event Objects (cont.)

- CN1 does not have different type of event objects as in Java (e.g. **`ActionEvent, MouseEvent, KeyEvent,`** etc.)

- Activating a component (e.g., pushing a button), using a key (pressing, releasing), or use of pointer (pressing, releasing, dragging, etc.) ALL produce an object of type **`ActionEvent`**.

Event Listeners

- Event-driven code attaches *listeners* to *event-generators*
- Event-generators make *call-backs* to listeners



ActionListener Interface

- **Listeners must implement interface `ActionListener` (build-in in CN1):**

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

Approaches for Creating a Listener

- (1) Have a class that implements **`ActionListener`**. Two options:
  - (1a) Your listener is different than the class that creates the components
  - (1b) You make the class that creates components (e.g., the class that extends **`Form`**) your listener
- (2) Have a class that extends build-in **`Command`** class. This approach uses the Command design pattern.

Approach (1a)

```
import com.codename1.ui.events.ActionEvent;
import com.codename1.ui.events.ActionListener;

/** 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 evt) {
    // we get here because the object being observed
    // generated an Action Event
   System.out.println ("Button Pushed...");
  }

}
```

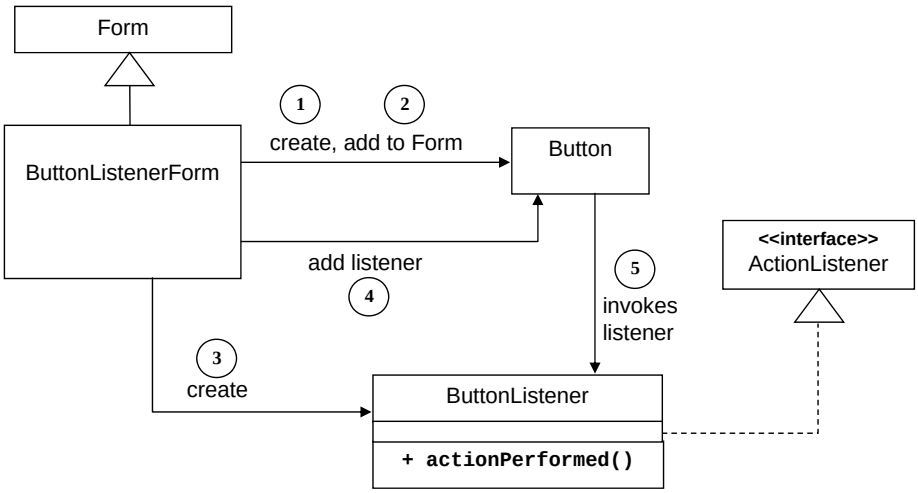
Using the Listener

Inside a class that extends from Form:

```
/** Code for a form ((ButtonListenerForm) with a single Button to which is attached an
 *  ActionListener. The button action listener is invoked whenever the
 *  button is pushed.
 */
//create a button
Button myButton = new Button("Button");
//…[style the button and add it to the form]
//create a separate ActionListener for the button
ButtonListener myButtonListener = new ButtonListener ();
//register the myButtonListener as an Action Listener for
//action events from the button
myButton.addActionListener(myButtonListener);
```

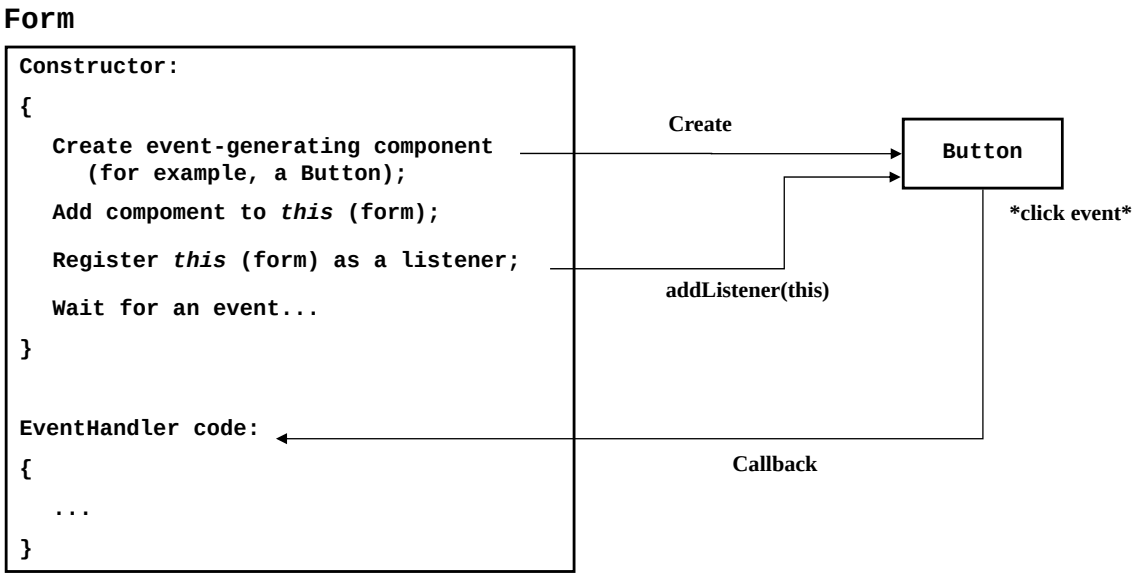

Listener Class Organization

- UML for the previous code:



Approach (1b)

Forms can listen to their own components!

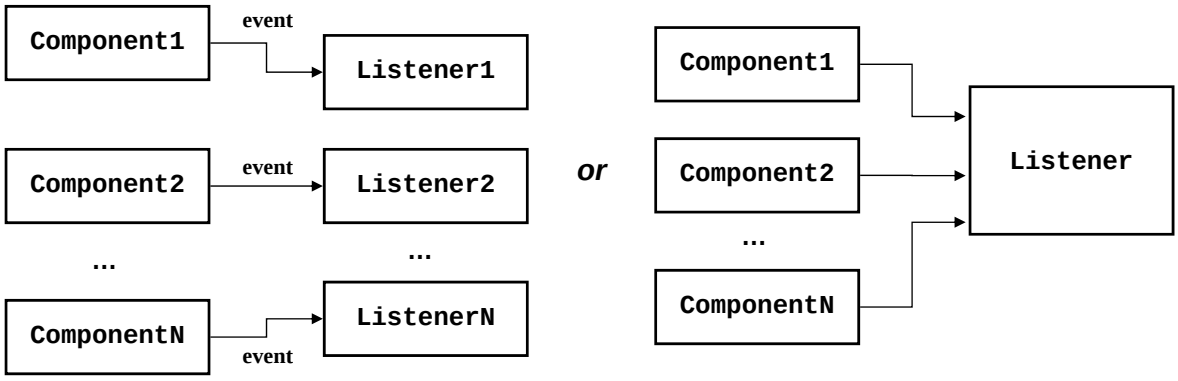


ActionListener Form Example

```
/** Code for a form with a single button which the form listens to. */
public class SelfListenerForm extends Form implements ActionListener {
  public SelfListenerForm () {
     // create a new button
     Button myButton = new Button ("Button");
     // add the button to the content pane of this form
     add(myButton);
     // register THIS object (the form) as an Action Listener for
     //  action events from the button
     myButton.addActionListener(this);
     show();
  }
  // Action Listener method: called from the button because
  // this object -- the form -- is an action listener for the button
  public void actionPerformed (ActionEvent e) {
     System.out.println ("Button Pushed (printed from the form)...");
  }
}
```

Multiple Event Sources

- *Approaches:*
  - (1a) requires multiple separate listeners
  - (1b) requires one listener
    - it would need to be able to *distinguish event source*



Let's consider this second option …

Multiple Component Listener

```
/* Code for a form with multiple buttons which have action handlers in the form */
public class MultipleComponentListener extends Form implements ActionListener{
  private Button buttonOne = new Button("Button One"); //need to make this button a class field
  public MultipleComponentListener() {
    setTitle("Multiple Component Listener");
    Button buttonTwo = new Button("Button Two");
    //...[set styles of the buttons and add them to form]
    buttonOne.addActionListener(this);
    buttonTwo.addActionListener(this);
    show();
   }
  public void actionPerformed(ActionEvent evt) {
    if(evt.getComponent().equals(buttonOne)){ //buttonOne must be a class field
      System.out.println ("Button One Pushed (printed from the form using
                                                          getComponent())...");
    }
    else if(((Button)evt.getComponent()).getText().equals("Button Two")){
      //if we change the label of the button, this code would not work
      System.out.println ("Button Two Pushed (printed from the form using
                                                  getComponent().getText())...");
    }//else if
  }//actionPerformed
}//class
```

Multiple Component Listener (cont.)

- ***actionPerformed()*** would get bigger and bigger… more and more unwieldy as we have more components in the form.
- A better approach is using combination of approaches (1a) and (1b):

Command Design Pattern

which is the Approach (2).

(use one listener for all related components, but you can have multiple listeners for different groups of components)

Anonymous `Command` Sub-Class

We can extend from **`Command`** in a separate .java file and then instantiate an object of this sub-class in a separate .java file.

Or… we generate an object of an anonymous sub-class of **`Command`** in the same .java file.

First option (which is used in the “Command Design Pattern” code example) is recommended…

See the next slide for the second option…But do **NOT use** the second approach (**anonymous sub-classing**) in the assignments!

Anonymous Command Sub-Class (cont.)

```
/* Code for a form that creates an object of anonymous sub-class of the Command */
//create a Toolbar called myToolBar and add it to the form
//create the object (called inforTitleBarAreaItem) of anonymous sub-class of Command
Command infoTitleBarAreaItem = new Command("Info") {
      public void actionPerformed(ActionEvent ev) {
            String Message = "I provide information.";
            Dialog.show("Info", Message, "Ok", null);
       }
};
myToolbar.addCommandToRightBar(infoTitleBarAreaItem);
```


Adding a Command to Side Menu Component

```
/* Code for a form which has a CheckBox as a side menu item*/
public class SideMenuItemCheckForm extends Form{
  private Label checkStatusVal = new Label("OFF");
  public SideMenuItemCheckForm() {
  //...[add a Toolbar and some side menu items]
  CheckBox checkSideMenuComp = new CheckBox("Check Side Menu Component");
  //...[change style of the check box]
  //create a command object and set it as the command of check box
  Command mySideMenuItemCheck = new SideMenuItemCheck(this);
  checkSideMenuComp.setCommand(mySideMenuItemCheck);
  //set "SideComponent" property of the command object to the check box
  mySideMenuItemCheck.putClientProperty("SideComponent", checkSideMenuComp);
  //add the command to the side menu, this places its side component (check box) in the side menu
  myToolbar.addCommandToSideMenu(mySideMenuItemCheck);
  //add a label to indicate the check box value on the form, divide the label to two parts, text
  //and value, and add padding to value part so that the labels looks stable when value changes
  Label checkStatusText = new Label("Check Box Status:");
  checkStatusVal.getAllStyles().setPadding(LEFT, 5);
  checkStatusVal.getAllStyles().setPadding(RIGHT,5);
  //...[add labels to the form and show the form]
  }
```

continued...

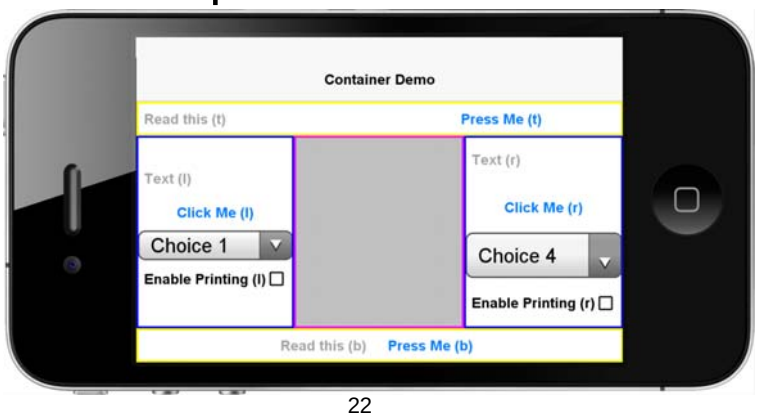
Adding a Command to Side Menu Component

continued…

```
public void setCheckStatusVal(boolean bVal){
  if (bVal)
    checkStatusVal.setText("ON");
  else
   checkStatusVal.setText("OFF");} //call repaint(), if cannot see values properly
}// SideMenuItemCheckForm class   ---------- below is the code for the command class
public class SideMenuItemCheck extends Command {
  private SideMenuItemCheckForm myForm;
  public SideMenuItemCheck (SideMenuItemCheckForm fForm){
    super("Side Menu Item Check"); //do not forget to set the "command name"
    myForm = fForm;}
@Override
public void actionPerformed(ActionEvent evt){
  if (((CheckBox)evt.getComponent()).isSelected())//getComponent() returns the component
                                                  //that generated the event
    myForm.setCheckStatusVal(true);
  else
    myForm.setCheckStatusVal(false);
  SideMenuBar.closeCurrentMenu(); //do not forget to close the side menu
  }//actionPerformed
}// SideMenuItemCheck class
```


Component Width and Height

- Layout managers automatically place and size the components.
- Hence, we can only get their correct width and height values after calling **`show()`**.
- Remember the "Container Example" from the "GUI Basics" chapter:



Component Width and Height (cont.)

```
public class FormWithMultipleContainers extends Form{
  Container centerContainer;
  public FormWithMultipleContainers(){
    //create the center container and add it to form
   centerContainer = new Container();
    //... [add the centerContainer to the from, create bottomContainer]
    //create a button and add it to bottomContainer
   Button bPressMeB = new Button("Press Me (b)");
   bottomContainer.add(bPressMeB);
   //...[add the bottom Container to the from,
   //create/add other containers and components and style them all]
   //below line prints incorrect values: 0,0
   System.out.println("Center container width/height (printed BEFORE show()):
      " + centerContainer.getWidth() + " " + centerContainer.getHeight());
   show();
    //below line prints correct width and height
   System.out.println("Center container width/height (printed AFTER show()): "
       + centerContainer.getWidth() + " " + centerContainer.getHeight());
   bPressMeB.addActionListener(new Command("Print center"){
         public void actionPerformed(ActionEvent ev){
         //below line also prints correct width and height
         System.out.println("Center container width/height (printed after
                  button click): " + centerContainer.getWidth() + " " +
                  centerContainer.getHeight());
         }//actionperformed(){
         }//new Command(){
         );//addActionListener(
         }//constructor
}//class
```


Pointer Handling

- Components also generate an **`ActionEvent`** when a pointer is pressed/released or dragged on them.
- **`Component`** class provides:

  `addPointerPressedListener( )`

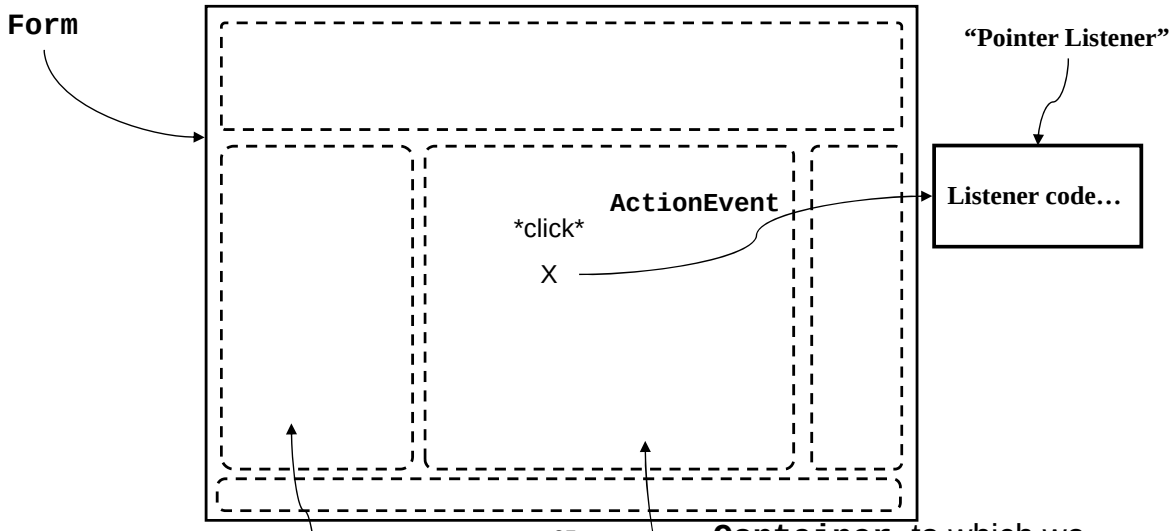
  `addPointerReleasedListener( )`

  `addPointerDraggedListener( )`

…all of which take a parameter of **`ActionListener`** ...

(this means you can attach a **`Command`** and pointer actions can also become a part of Command Design Pattern)

Pointer Handling (cont.)



`Container` without a listener (clicking on this container would not invoke any action/method)

`Container` to which we have added a listener (clicking on this container would call **`actionPerformed()`** of the attached listener)

Pointer Handling (cont.)

- Like action listeners, pointer listeners must also implement **`ActionListener`** interface:

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

- **`ActionEvent`** passed to **`actionPerformed()`** method has **`getX()`** and **`getY()`** methods which returns the "screen coordinate" of the pointer location.

Pointer Listener Example

```
/** A Form with a simple pointer-responding container */
public class PointerListenerForm extends Form{
  public PointerListenerForm() {
  //...[set the form layout to borderlayout, generate and style buttons and
  //add them to on north and south containers]
  //have an empty container in the center and add a pointer pressed
  //listener to it
  Container myContainer = new Container();
  PointerListener myPointerListener = new PointerListener ();
  myContainer.addPointerPressedListener(myPointerListener);
  this.add(BorderLayout.CENTER,myContainer);
  //…[add other containers and components to the form]
  }
}
--------
public class PointerListener implements ActionListener {
  public void actionPerformed(ActionEvent evt) {
   System.out.println("Pointer x and y: " + evt.getX() + " " + evt.getY());
  }
}
```


Pointer Listener Example

Question:

What happens if I add the listener to the form instead of the container in the form?

```
public class PointerListenerForm extends Form{
  public PointerListenerForm() {
    PointerListener mypointerListener = new PointerListener();
    this.addPointerPressedListener(mypointerListener);
    //…[add containers and components to the form]
 }
}
```

Answer:

Clicking anywhere on the form (including the title bar area) would print out the values…

Adding Listeners for Different Pointer Actions

- There are two approaches:
  - You can add a separate listener for pressed/released/dragged

```
myContainer.addPointerPressedListener(myPressedListener)
myContainer.addPointerReleasedListener(myReleasedListener)
myContainer.addPointerDraggedListener(myDraggedListener)
```

    - This approach requires us to have three separate listener classes.
  - You can have a single listener for all (e.g., self listener) and distinguish between different actions by using **`ActionEvent`**'s **`getEventType()`** method.
    - You need to have if-then-else structure which can get unwieldy if the form is also listening for other event types

Adding Pointer Listener vs Overriding Pointer Methods

- **`Component`** class also has following methods:

  `pointerPressed()`

  `pointerReleased()`

  `pointerDragged()`

  ….all of which gets the parameters which indicate screen location of the pointer…

- If you are extending from a **`Component`** (e.g. **`Form, Container`**), you can override these functions. This is the recommended approach since it is easier than adding a listener for each separate pointer action.

Overriding Pointer Methods

```
/* Center container of the form is a PointerContainer which extends from Container */
public class PointerListenerForm extends Form{
  public PointerListenerForm() {
    PointerContainer myPointerContainer = new PointerContainer();
    this.add(BorderLayout.CENTER,myPointerContainer);
    //…[add other containers and components to the form]   }
}
------
/* We can override the pointer methods in the Container */
public class PointerContainer extends Container{
  @Override
  public void pointerPressed(int x,int y){
    System.out.println("Pointer PRESSED x and y: " + x + " " + y);   }
  @Override
  public void pointerReleased(int x,int y){
    System.out.println("Pointer RELEASED x and y: " + x + " " + y);  }
  @Override
   public void pointerDragged(int x,int y){
     System.out.println("Pointer DRAGGED x and y: " + x + " " + y);  }
}
```

10 - Interactive Techniques

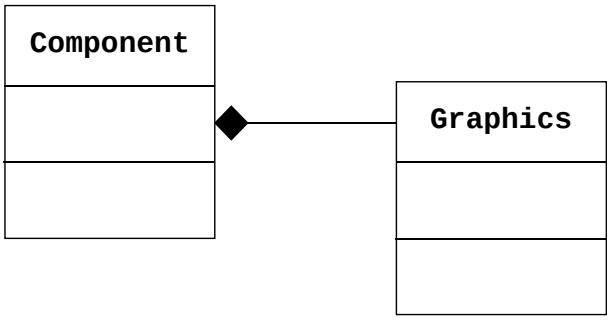
Computer Science Department
California State University, Sacramento

Overview

- **`Graphics` Class (and object)**
- **Component Repainting, `paint()`**
- **Graphics State Saving**
- **Onscreen Object Selection**

Component Graphics

- **Every `Component` contains an object of type `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);`**
  - **`drawArc (int x, int y,  int width, int height, int startAngle, int arcAngle);`**

    e.g., to draw a filled circle with radius r:

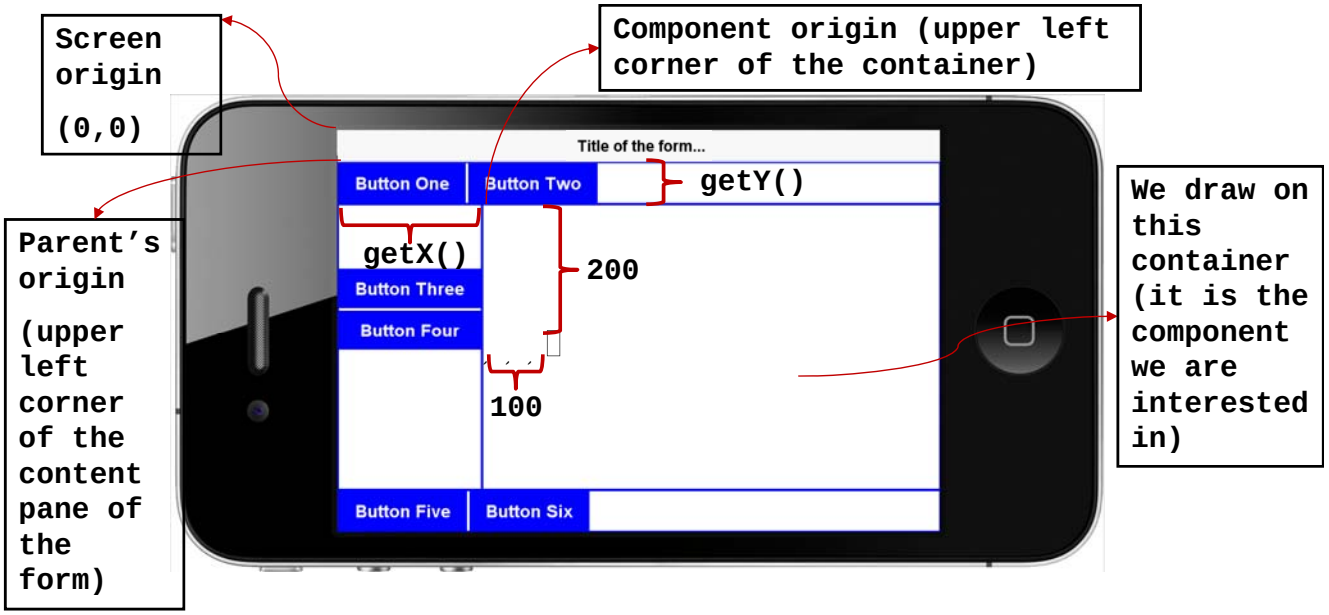
    `fillArc(x, y, 2*r, 2*r, 0, 360);`
  - **`drawPolygon(int[] xPoints, int[] yPoints, int nPoints)`**

    e.g., you can draw a triangle using the drawPolygon()...
  - **`drawString (String str,  int x,  int y);`**
  - **`setColor (int RGB);`**
  - **`. . .`**

Drawing Coordinates

- Drawing coordinates (e.g. x/y in **`drawLine()`**) indicate the location of upper left corner of the shape that is being drawn.
- Drawing coordinates are **relative to the component's parent's "origin" (not the component's origin … it's parent's origin)**
- Parent is the container that holds the component. If we add a component (e.g. container) to a form, content pane of the form would be the parent of the component.
- Origin of the parent/component is at its upper left corner.
- **`getX()/getY()`** methods of **`Component`** return the component's origin location relatively to its parent's origin location.

Drawing Coordinates (cont.)



So to draw a rectangle at 100 pixels right and 200 pixels down of the origin of the component:

```
drawRect(getX()+100, getY()+200, width, height)
```

Getting a reference to the `Graphics` object

- But how can we get a hold of **`Graphics`** object of a component to call the draw methods on it??
- "Component repainting" mechanism allows us to get a hold of this reference…

Component Repainting

- **Every `Component` has a `repaint()` method**
  - Tells a component to update its screen appearance
  - Called automatically whenever the component needs redrawing
    - e.g., app is opened for the first time, user switched back to the app while multi-tasking among different apps, a method such as **`setBgColor(int RGB)`** is called…
  - Can also be called manually by the application code to force a redraw

Component Repainting (cont.)

- **`Component`** also contain a method named **`paint()`**
  - **`repaint()` passes the `Graphics` object to the component's `paint()` method**
  - **`paint()`** is responsible for the actual drawing (using **`Graphics`**)
  - Never invoke **`paint()`** directly; always call **`repaint()`** since repaint() does other important operations…

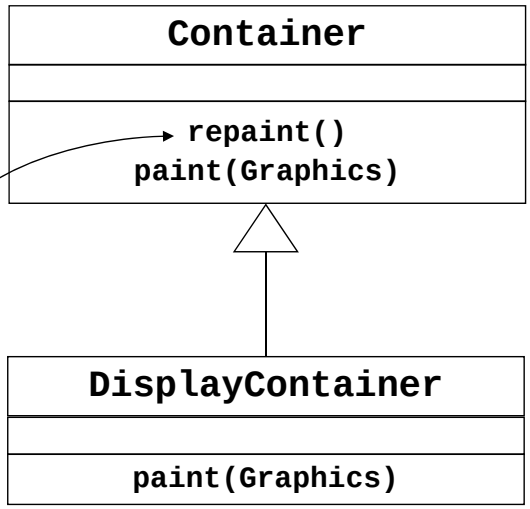
Differences between Java and CN1

- Java AWT/Swing component has **`getGraphics()`** method which returns **`Graphics`** object of the component.
- CN1 UI component does not have this method....
- Only way to get a hold of **`Graphics`** object is through overriding **`paint()`** method.

Overriding paint()

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

```
public class MyForm extends Form {
   private DisplayContainer myContainer;
   public MyForm() {
      ...
      myContainer = new DisplayContainer();
      ...
   }
   public void someMethod() {
      ...
      myContainer.repaint();
      ...
   }
}
```



Overriding paint()(cont.)

- Always perform the drawing in the overriden **paint()** method.
  - Never save the **Graphics** object and use it in another method to draw things! If you do so:
    - Drawn things would vanish the next time **repaint()** is called …
    - Drawn things would be located in wrong positions…
- The first line of the overriden **paint()** method **must** be **super.paint()!**
  - default **paint()** method performs other important operations necessary for updating component's screen appearance…

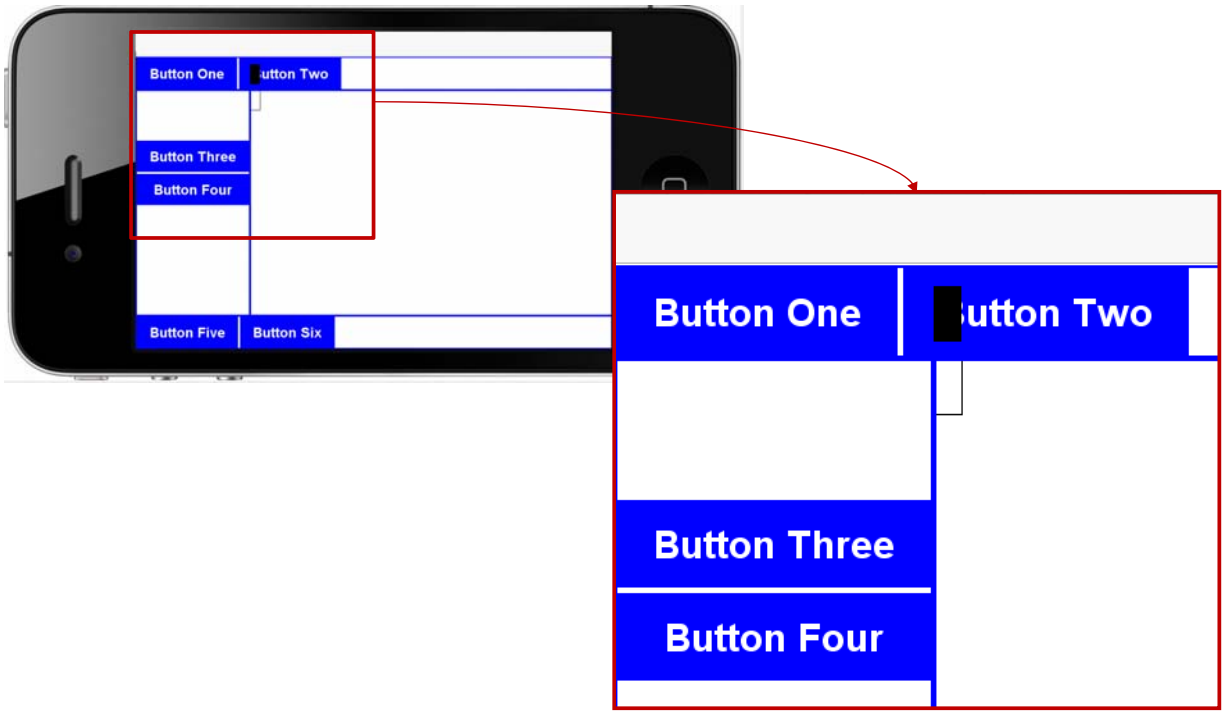
Non-working example

```
public class NonWorkingGraphics extends Form implements ActionListener{
CustomContainer myCustomContainer = new CustomContainer();
 public NonWorkingGraphics() {
  //... [use border layout and add north, east, south containers (each
  //include two styled buttons)]
  buttonOne.addActionListener(this);
  this.add(BorderLayout.CENTER, myCustomContainer);
 }
 public void actionPerformed(ActionEvent evt) {
  myCustomContainer.drawObj();
 }
}
```


Non-working example (cont.)

```
public class CustomContainer extends Container{
 private Graphics myGraphics;
 public void paint(Graphics g){
  myGraphics = g;
  super.paint(g);
  myGraphics.setColor(ColorUtil.BLACK);
  //empty rectangle appears in the CORRECT place (at the origin of this)
  myGraphics.drawRect(getX(), getY(), 20, 40);
 }
 public void drawObj(){
  repaint();
  myGraphics.setColor(ColorUtil.BLACK);
  //filled rectangle appears in the WRONG place and disappears next time
  //repaint() is called
  myGraphics.fillRect(getX(), getY(), 20, 40);
 }
}
```

Non-working example (cont.)



Importance of getX()/getY()

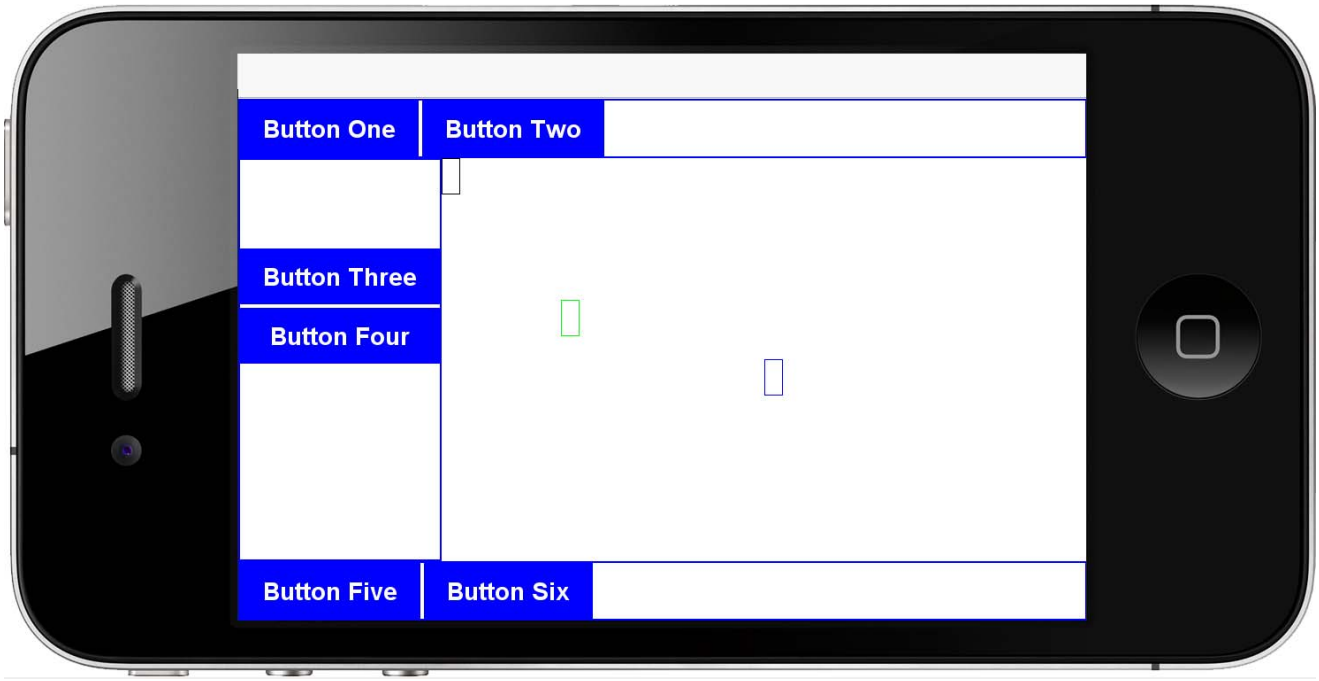
Assume we would like to draw a rectangle in the middle of **`CustomContainer.`**

If we have the following **`paint()`** method:

```
public void paint(Graphics g){
 super.paint(g);
 int w = getWidth();
 int h = getHeight();
 g.setColor(ColorUtil.BLACK);
 g.drawRect(getX(), getY(), 20, 40);
 g.setColor(ColorUtil.GREEN);
 g.drawRect(w/2, h/2, 20, 40);
 g.setColor(ColorUtil.BLUE);
 g.drawRect(getX()+w/2, getY()+h/2, 20, 40);
}
```

Importance of getX()/getY() (cont.)

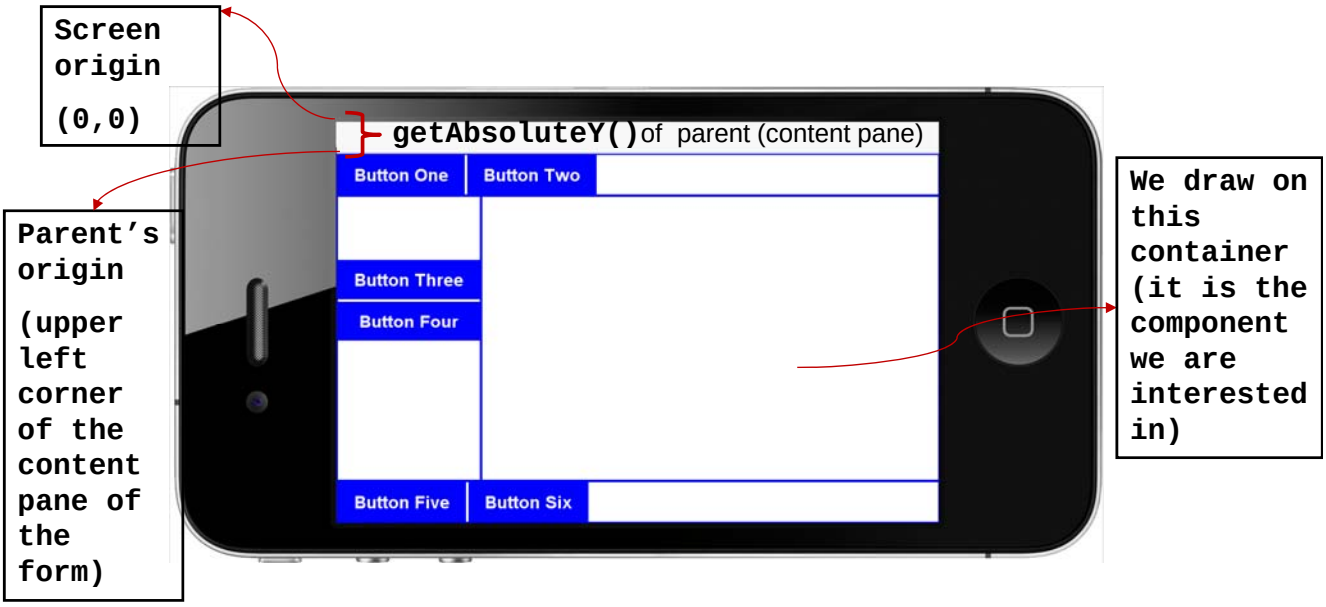
Only the blue rectangle would appear in the center of the **`CustomContainer`**…



Pointer Graphics

- We would like to draw a rectangle where ever the user presses on the **`CustomContainer`**.
- Pointer pressed gets coordinates relative to the screen origin (upper left corner of the screen).
- However draw methods expects coordinates relative to the component's parent's origin.
- You can convert screen coordinate to parent coordinate using **`getAbsoluteX()` and `getAbsoluteY()`** methods of the parent container.
- You can get the parent using **`getParent()`** method of the component.

Pointer Graphics (cont.)



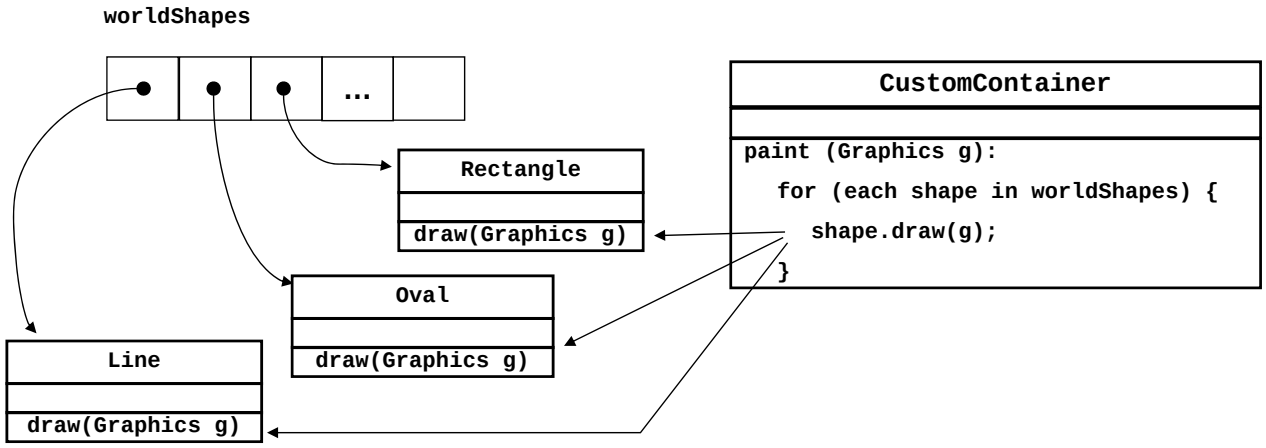
`getAbsoluteX()` of parent (content pane) is 0 in this example…

Pointer Graphics Example

```
public class CustomContainer extends Container{

  private int iPx = 0;
  private int iPy = 0;

  @Override
  public void paint(Graphics g){
    super.paint(g);
    g.setColor(ColorUtil.BLACK);
    //make the point location relative to the component's parent's origin
    //and then draw the rectangle (below un-filled rect would appear in the CORRECT location)
    g.drawRect(iPx-getParent().getAbsoluteX(),iPy-getParent().getAbsoluteY(),20,40);
    //below filled rect would appear in the WRONG location
    g.fillRect(iPx,iPy, 20,40);
  }

  @Override
  public void pointerPressed(int x,int y){
    //save the pointer pressed location
    //it is relative the to the screen origin
    iPx = x;
    iPy = y;
    repaint();
  }
}
```

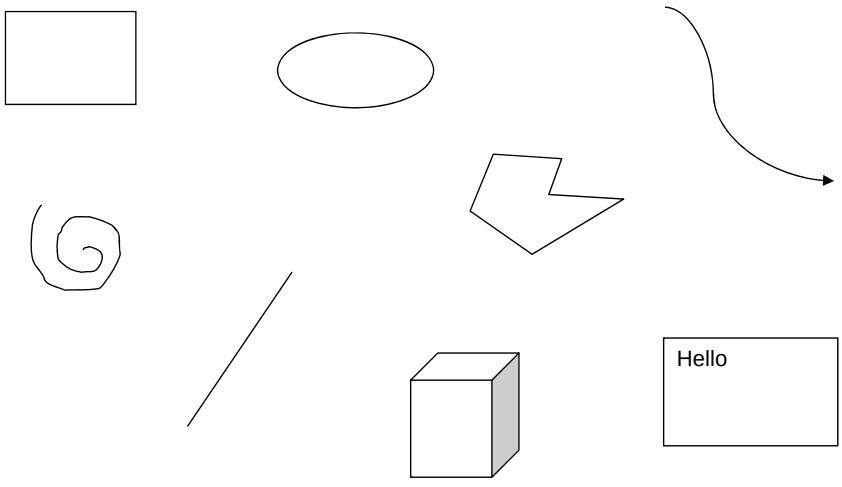
Maintaining Graphical State

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



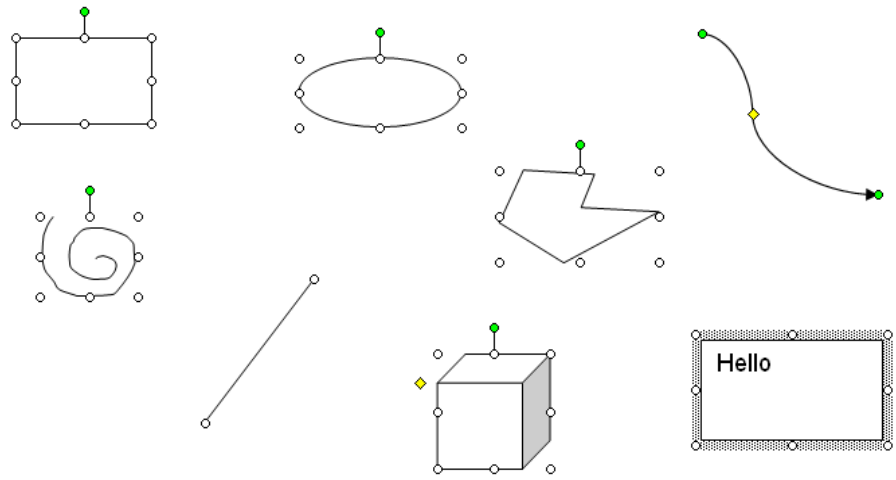
Object Selection

- Various *unselected* objects:



Object Selection (cont.)

- *Selected* versions of the same objects:

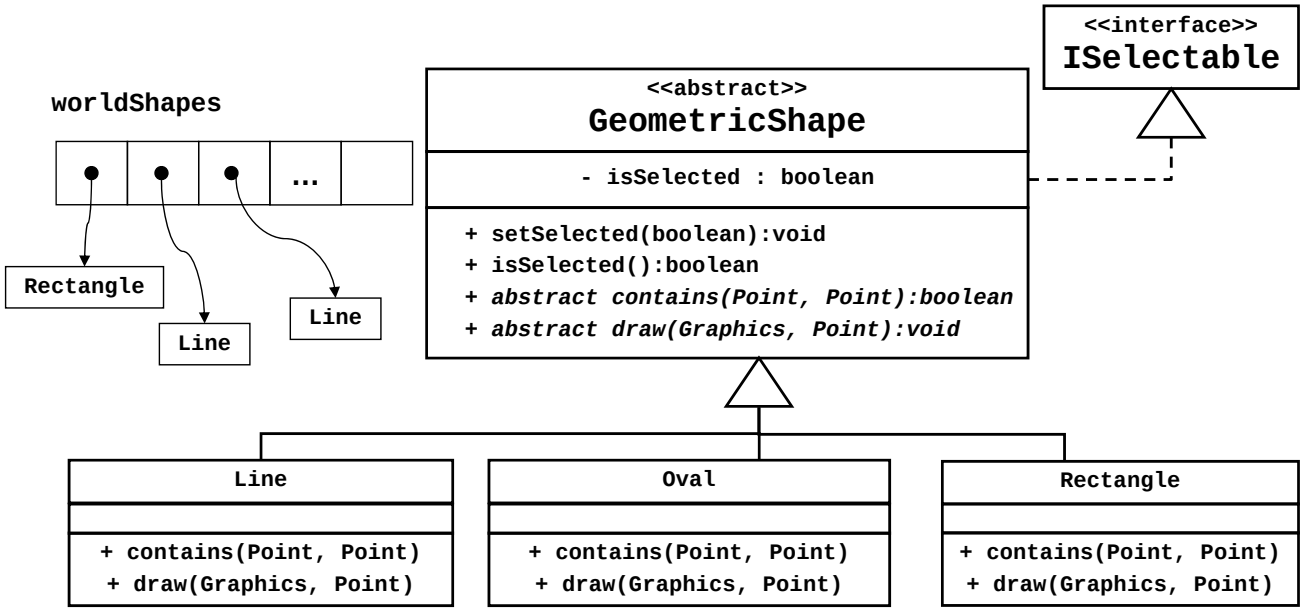


Defining "Selectability"

```
/** 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 pointer is "in" an object
 // pPtrRelPrnt is pointer position relative to the parent origin
 // pCmpRelPrnt is the component position relative to the parent origin
 public boolean contains(Point pPtrRelPrnt, Point pCmpRelPrnt);

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

Implementing Object Selection

(1) Expand objects to support selection



Implementing Object Selection (cont.)

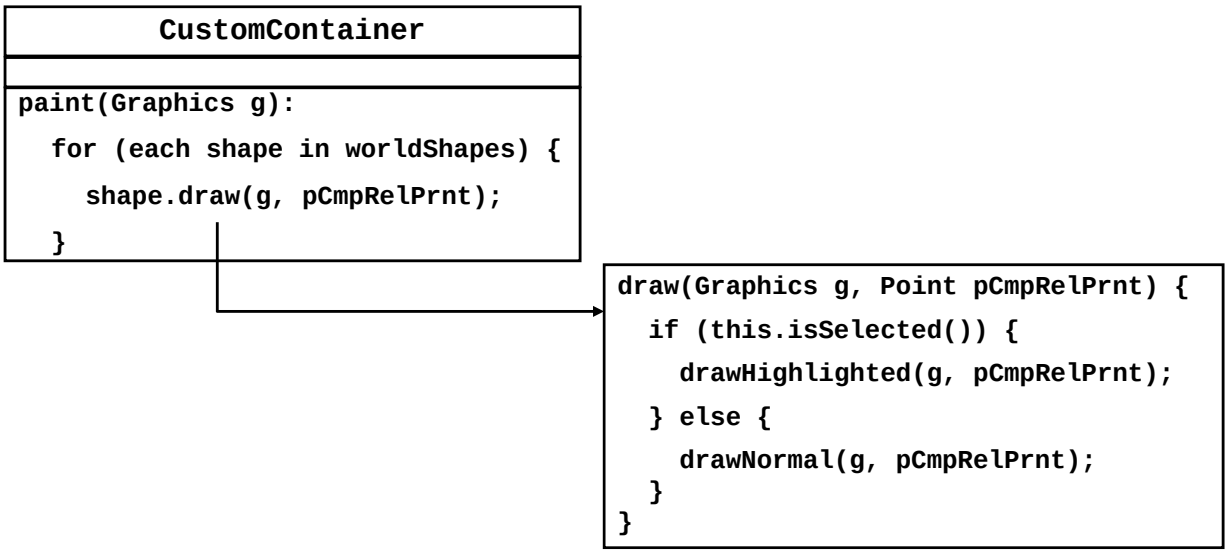
(2) On pointer pressed:

- Determine if pointer is “inside” any shape
  - if shape contains pointer, mark as “selected”
- Repaint container

```
//overriding pointerPressed() in CustomContainer
import com.codename1.ui.geom.Point;
void pointerPressed(int x, int y) {
//make pointer location relative to parent's origin
  x = x - getParent().getAbsoluteX();
  y = y - getParent().getAbsoluteY();
  Point pPtrRelPrnt = new Point(x, y);
  Point pCmpRelPrnt = new Point(getX(), getY());
  for (each shape in worldShapes) {
    if (shape.contains(pPtrRelPrnt, pCmpRelPrnt)) {
      shape.setSelected(true);
    } else {
      shape.setSelected(false);
    }
  }
  repaint();
}
```

Implementing Object Selection (cont.)

(3) Draw "selected" objects in different form



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, Point pCmpRelPrnt);
  abstract boolean contains(Point pPtrRelPrnt, Point pCmpRelPrnt);
}
```

```
public class MyRect extends GeometricShape {
  //...[assign iShapeX and iShapeY to rect coordinates (upper left corner of rect
  //which is relative to the origin of the component) supplied in the constructor]
  public boolean contains(Point pPtrRelPrnt, Point pCmpRelPrnt) {
    int px = pPtrRelPrnt.getX(); // pointer location relative to
    int py = pPtrRelPrnt.getY(); // parent's origin
    int xLoc = pCmpRelPrnt.getX()+ iShapeX;// shape location relative
    int yLoc = pCmpRelPrnt.getY()+ iShapeY;// to parent's origin
    if ( (px >= xLoc) && (px <= xLoc+width)
      && (py >= yLoc) && (py <= yLoc+height) )
      return true; else return false;}
  public void draw(Graphics g, Point pCmpRelPrnt) {
    int xLoc = pCmpRelPrnt.getX()+ iShapeX;// shape location relative
    int yLoc = pCmpRelPrnt.getY()+ iShapeY;// to parent's origin
    if(isSelected())
      g.fillRect(xLoc, yLoc, width, height);
    else
      g.drawRect(xLoc, yLoc, width, height);}
}
```

```
public class ObjectSelectionForm extends Form {
  private Vector<GeometricShape> worldShapes = new Vector<GeometricShape>();
  public ObjectSelectionFrame() {
  // ...code here to initialize the form with a CustomContainer...
  //specify rect coordinates (relative to the origin of component), size, and color
    worldShapes.addElement(new MyRect(100, 100, 50, 50, ColorUtil.BLACK));
    worldShapes.addElement(new MyRect(200, 200, 100, 100, ColorUtil.GREEN));}
}
```

```
public class CustomContainer extends Container {
  //…assume we pass worldShapes to the constructor of CustomContainer
  public void paint(Graphics g) {
      super.paint(g);
      Point pCmpRelPrnt = new Point(getX(), getY());
      for(int i=0; i<worldShapes.size();i++)
          worldShapes.elementAt(i).draw(g, pCmpRelPrnt);}
  public void pointerPressed(int x, int y) {
      x = x - getParent().getAbsoluteX();
      y = y - getParent().getAbsoluteY();
      Point pPtrRelPrnt = new Point(x, y);
      Point pCmpRelPrnt = new Point(getX(), getY());
      for(int i=0;i<worldShapes.size();i++) {
        if(worldShapes.elementAt(i).contains(pPtrRelPrnt, pCmpRelPrnt))
          worldShapes.elementAt(i).setSelected(true);
        else
          worldShapes.elementAt(i).setSelected(false);
      }
      repaint(); }
}
```

11 - Introduction to Animation

Computer Science Department
California State University, Sacramento

Overview

- **Frame-based Animation**
- **Timers**
- **Moving Images**
- **Self-drawing and Self-animating Objects**
- **Computing Animated Location**
- **Collision Detection and Response**

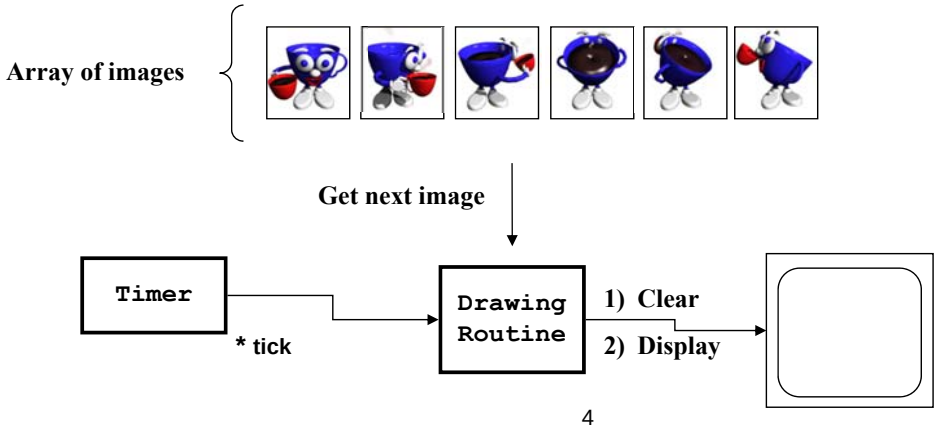
Frame-Based Animation

- Similar images shown in rapid succession imply movement

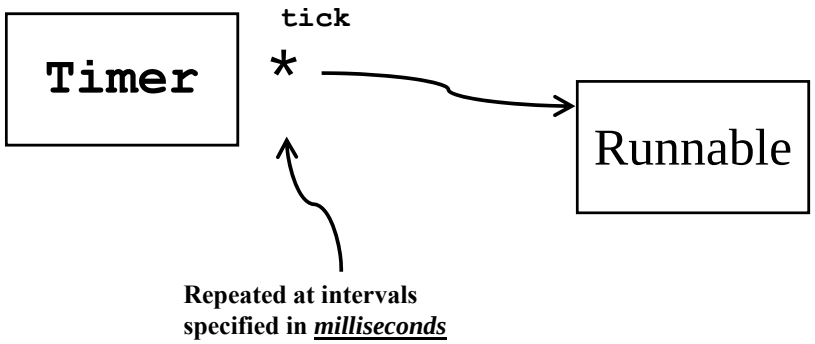
Image credit: Graphic Java: Mastering the JFC (3rd ed.), David Geary

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



CN1 `UITimer` Class



CN1 `UITimer` Class (cont)

- Its constructor accepts a runnable to invoke on each tick: `UITimer(Runnable r)`
- It must be linked to a specific form:

```
schedule(int timeMillis, boolean repeat, Form bound)
```

- It is invoked on the CodenameOne main thread rather than on a separate thread.
- It is different from Java Swing **`Timer`** which generates action events in every tick…
- No need to start the timer (**`schedule()`** starts it), use **`cancel()`** to stop it.

CN1 `UITimer` Class (cont)

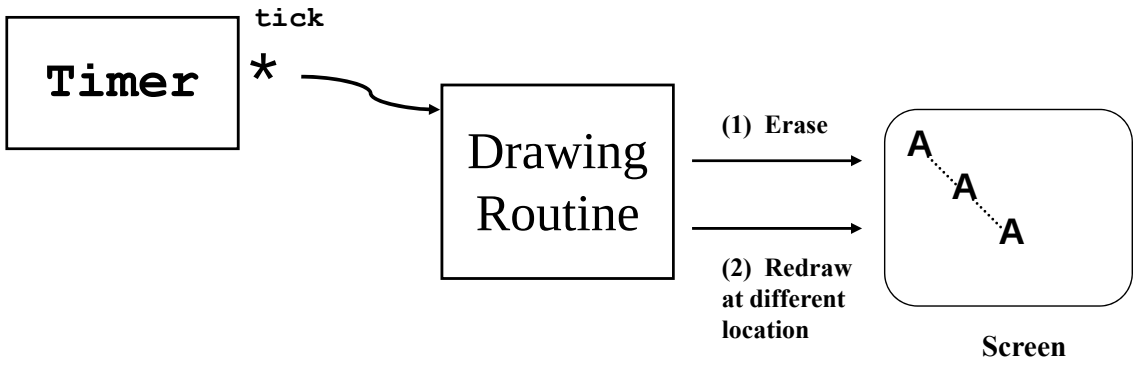
- **Runnable attached to the timer must implement interface *Runnable* (built-in CN1 interface):**

```
interface Runnable
        {
                public void run ();
        }
```


Using the UITimer

```
/** This class creates and binds the Timer to the form and provides a runnable (which is
 * the form itself) for the Timer.  The runnable draws graphical shapes of random sizes at
 * random locations. */
public class TimerGraphics extends Form implements Runnable {
  private TimerGraphicsContainer myContainer;
  public TimerGraphics() {
    // ...code here to initialize the form which uses border layout...
    // create a container on which to do graphics; put it in the center
    myContainer = new TimerGraphicsContainer();
    add(BorderLayout.CENTER, myContainer);
    //create timer and provide a runnable (which is this form)
    UITimer timer = new UITimer(this);
    //make the timer tick every second and bind it to this form
    timer.schedule(1000, true, this);}
    // Entered when the Timer ticks
  public void run() {
    myContainer.repaint();}
}
public class TimerGraphicsContainer extends Container{
  public void paint(Graphics g){
    super.paint();
    g.setColor(ColorUtil.BLACK);
    int iShapeX = myRNG.nextInt(getWidth()); //shape location (relative to the
    int iShapeY = myRNG.nextInt(getHeight());//the origin of the container)
    int xSize = myRNG.nextInt (50);
    int ySize = myRNG.nextInt (25);
    //draw a random-sized rounded corner rectangle at a random location
    g.drawRoundRect(getX()+ iShapeX, getY()+ iShapeY,xSize,ySize,20,10);}
}
```

Animation via Image Movement

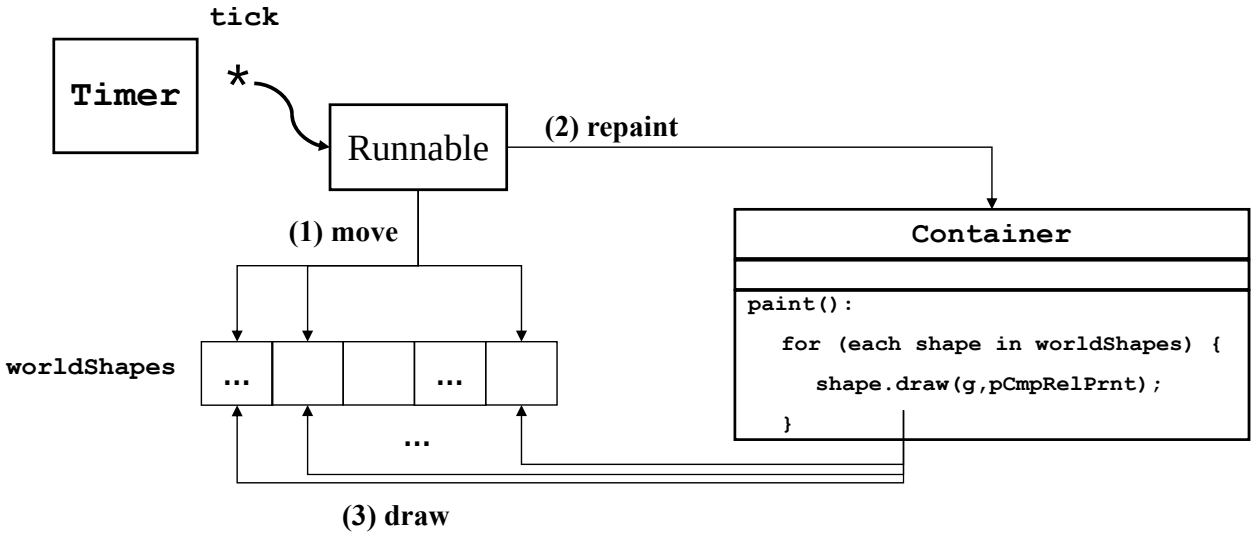


Animation Example

```
/*This time instead of drawing shapes of random sizes at random locations,
 * we will draw the same image (a simple filled shape) that moves on a path.
 * The form is the same as above example except that the tick would happen every 100 ms… */
public class AnimationContainer extends Container {
  private int currentX = 0, currentY = 0 ; // image location (relative to the origin
                                           //of the component)
  private int incX = 3, incY = 3 ;         // amount of movement
  private int size = 20 ;
  // update the image on the container
  public void paint(Graphics g) {
    super.paint (g);
    // draw the image (a simple filled rounded corner rect) at the current location.
    g.setColor(ColorUtil.BLACK);
    g.fillRoundRect(getX()+currentX, getY()+currentY, size, size, 20, 10);
    // update the image position for the next draw
    currentX += incX ;
    currentY += incY ;
    // reverse the movement direction if the image reaches an edge
    if ( (currentX+size >= getWidth()) || (currentX < 0) )
      incX = -incX ;
    if ( (currentY+size >= getHeight()) || (currentY < 0) )
      incY = -incY ;
  }
}
```

"Self-Animating" Objects

- Objects should be responsible for their own drawing and movement



"Self-Animation" Example

```
/** A form containing a collection of "self drawing objects". */
public class SelfDrawerAnimationForm extends Form implements Runnable {
  private SelfAnimationContainer myContainer ;
  public SelfDrawerAnimationFrame() {
    //...code here to initialize the frame with a BorderLayout
    // create a world containing a self-drawing object
    Vector<WorldObject> theWorld = new Vector<WorldObject>();
    theWorld.add( new WorldObject() );
    //create a container on which the world will be drawn
    myContainer = new SelfAnimationContainer(theWorld) ;
    add(BorderLayout.CENTER, myContainer);
    // create a Timer and schedule it
    UITimer timer = new UITimer (this);
    timer.schedule(15, true, this);
  }
  // called for each timer tick: tells object to move itself, then repaints the container
  public void run () {
    Dimension dCmpSize = new Dimension(myContainer.getWidth(),
                                       myContainer.getHeight());
    for (WorldObject obj : theWorld) {
      obj.move(dCmpSize);
    }
    myContainer.repaint();
}
```

"Self-Animation" Example (cont.)

```
/** This class defines an object which knows how to "move" itself, given a container
 *  with boundaries, and knows how to "draw" itself given a Graphics object and container
 *  coordinates relative to its parent.*/
public class WorldObject {
  private int currentX = 0, currentY = 0 ; // the object's current location (relative to
                                           // the origin of the component)
  private int incX = 3, incY = 3 ;         // amount of movement on each move
  private int size = 35 ;                  // object size

  // create the image to be used for this object
  Image theImage = null;
  public WorldObject(){
    try {// you should copy happyFace.png directly under the src directory
      theImage = Image.createImage("/happyFace.png");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
  // move this object within the specified boundaries
  public void move (Dimension dCmpSize) {
    // update the object position
    currentX += incX ;
    currentY += incY ;

    // reverse the next movement direction if the location has reached an edge
    if ( (currentX+size >= dCmpSize.getWidth()) || (currentX < 0) )
      incX = -incX ;
    if ( (currentY+size >= dCmpSize.getHeight()) || (currentY < 0) )
      incY = -incY ;
  }
```

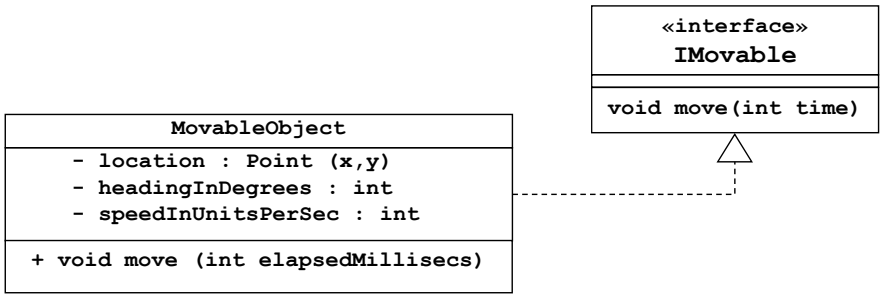
…continues…

"Self-Animation" Example (cont.)

```
  // draw the representation of this object using the received Graphics context
  public void draw(Graphics g, Point pCmpRelPrnt) {
    g.drawImage(theImage, pCmpRelPrnt.getX()+currentX,
                          pCmpRelPrnt.getY()+currentY, size, size);
  }
} //end of WorldObject class
-------------------------------------------------------------------------
/** A container which which redraws its world object(s) each time
 *  the container is repainted.
 */
public class SelfAnimationContainer extends Container {

  private Vector<WorldObject> theWorld ;
  public DisplayPanel (Vector<WorldObject> world) {
    theWorld = world ;
  }
  public void paint(Graphics g) {
    super.paint(g);
    Point pCmpRelPrnt = new Point(getX(),getY());
    for (WorldObject obj : theWorld) {
      obj.draw(g, pCmpRelPrnt) ;
  }
}
```

Computing Animated Location

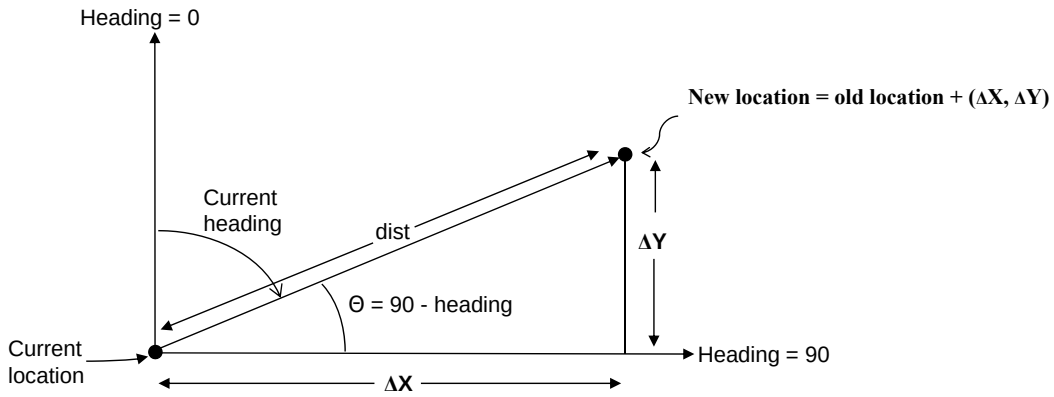
- Consider a "moveable object" defined as:



- 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:



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

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

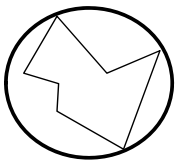
Collision Detection

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

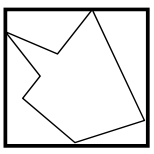
Collision Detection (cont.)

Simplification: "bounding volumes"

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



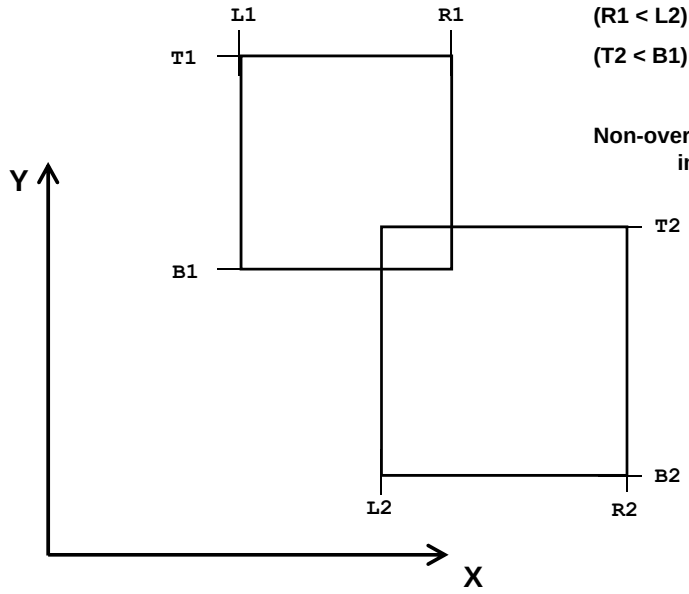
Bounding Circle



Bounding Rectangle

Collision Detection (cont.)

Bounding rectangle collisions



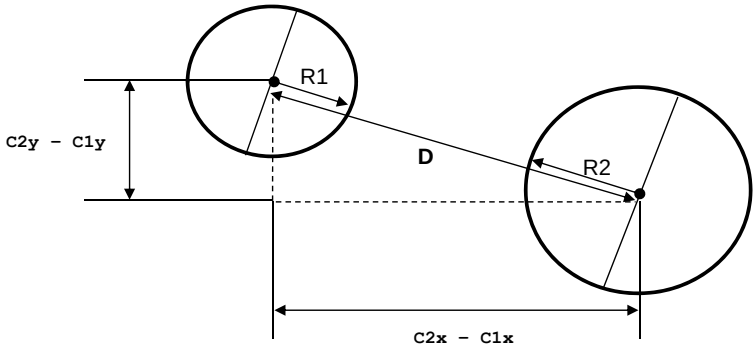
(R1 < L2) OR (L1 > R2) → No Left/Right overlap

(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.)

- **`ICollider` 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 form with self drawing objects. A Timer instructs the objects to move and
 *  a container to redraw the objects. On collision, an object changes color.  */
public class CollisionForm extends Form implements Runnable {
  private CollisionContainer myContainer;
  private Vector<RoundObject> theWorld ;

  public CollisionForm() {
    // code here to initialize the form...

    theWorld = new Vector<RoundObj>();

    // create a container on which the world objects will be drawn
    myContainer = CollisionContainer(theWorld) ;
    this.add(BorderLayout.CENTER,myContainer);
    // create a Timer to invoke move and repaint operations
    UITimer timer = new UITimer (this);
    timer.schedule(15, true, this);
    // create a world containing objects
    Dimension worldSize = new Dimension(myContainer.getWidth(),
                                        myContainer.getHeight());
    addObjects(worldSize);
  }

  private void addObjects(Dimension worldSize) {
    theWorld.addElement(new RoundObj(Color.red, worldSize));
    theWorld.addElement(new RoundObj(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 run () {
  // move all the world objects
  Iterator iter = theWorld.iterator();
  while(iter.hasNext()){
    ((IMovable) iter.next()).move();
  }
  // check if moving caused any collisions
  iter = theWorld.iterator();
  while(iter.hasNext()){
    ICollider curObj = (ICollider)iter.next(); // get a collidable object
    // check if this object collides with any OTHER object
    Iterator iter2 = theWorld.iterator();
    while(iter2.hasNext()){
      ICollider otherObj = (ICollider) iter2.next(); // get a collidable object
       // check for collision
      if(otherObj!=curObj){// make sure it's not the SAME object
        if(curObj.collidesWith(otherObj)){
          curObj.handleCollision(otherObj);
        }
      }
    }
  }
  myContainer.repaint(); // redraw the world
}
} //end class CollisionForm
```

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 RoundObj implements IMovable, IDrawable, ICollider {
  private static Random worldRNG = new Random();    // random number generator
  public void move () { ... }
  public void draw(Graphics g, Point pCmpRelPrnt) { ... }
  // Use bounding circles to determine whether this object has collided with another
  public boolean collidesWith(ICollider obj) {
    boolean result = false;
    int thisCenterX = this.xLoc + (objSize/2); // find centers
    int thisCenterY = this.yLoc + (objSize/2);
    int otherCenterX = obj.getX() + (objSize/2);
    int otherCenterY = obj.getY() + (objSize/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 = objSize/2;
    int otherRadius = objSize/2;
    int radiiSqr = (thisRadius*thisRadius + 2*thisRadius*otherRadius
                                          + otherRadius*otherRadius);
    if (distBetweenCentersSqr <= radiiSqr) { result = true ; }
    return result ;
  }
```

…continues…

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
      color = (ColorUtil.rgb(worldRnd.nextInt(256),
                             worldRnd.nextInt(256),
                             worldRnd.nextInt(256)));
    }
    // ...additional required interface methods here...
  } // end class RoundObject
  ------------------------------
  /** A container which redraws its object(s) each time it is repainted. */
  public class CollisionContainer extends Container {
    Vector<RoundObj> theWorld ;
    public CollisionContainer (Vector<RoundObj> aWorld) {
      theWorld = aWorld ;
    }
    public void paint (Graphics g) {
      super.paint(g);
      Point pCmpRelPrnt = new Point(getX(), getY());
      RoundObj next;
      Iterator iter = theWorld.iterator();
      while(iter.hasNext()){
        next = (RoundObj) iter.next();
        next.draw(g, pCmpRelPrnt);
      }
    }
  }
```

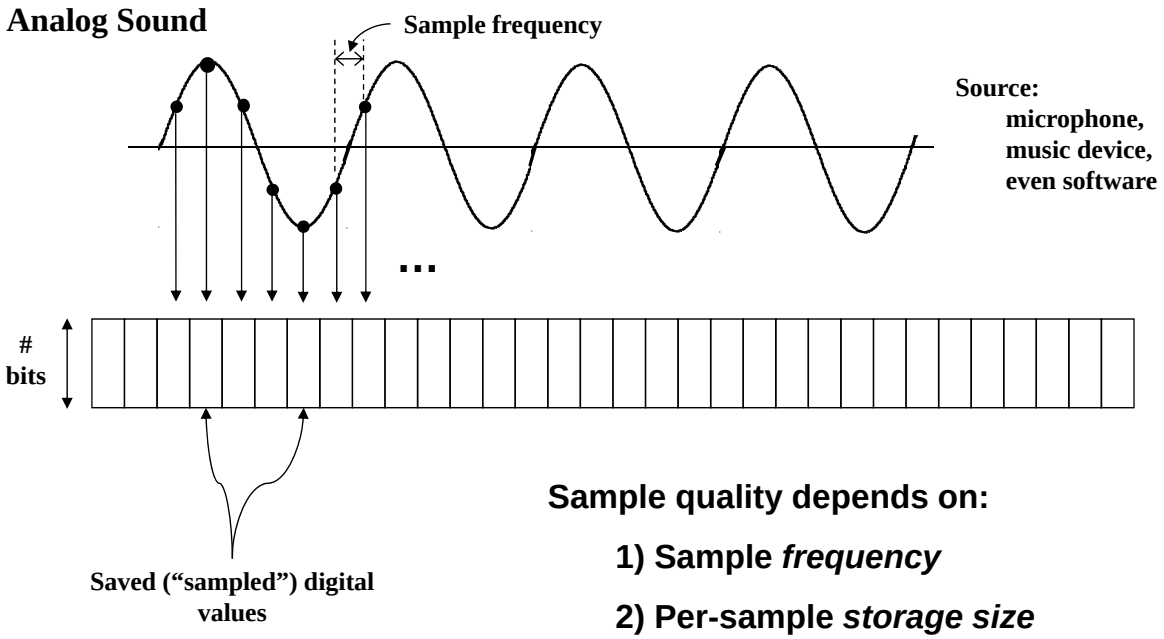
12 - Introduction to Sound

Computer Science Department
California State University, Sacramento

Overview

- **Sampled Audio**
- **Sound File Formats**
- **Popular Sound APIs**
- **Playing Sounds in CN1**
  - **Creating background sound that loops**

Sampled Audio



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

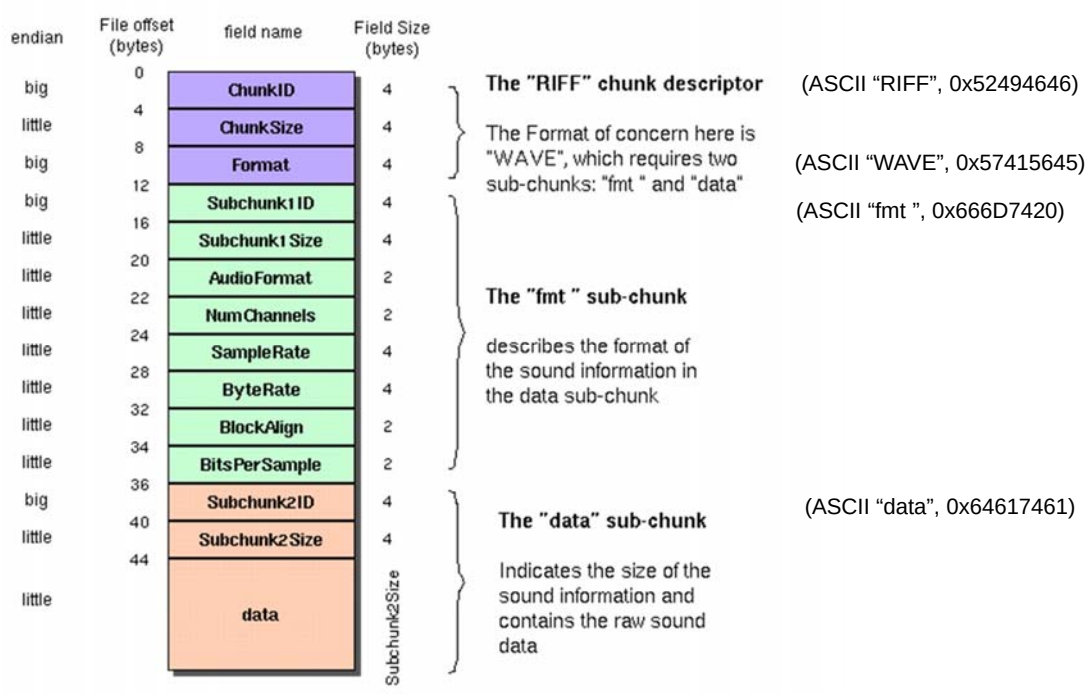


Image credit: http://ccrma.stanford.edu/courses/422/projects/WaveFormat/

Popular Sound API's

- Java **`AudioClip`** Interface
- JavaSound
- DirectSound / DirectSound3D
- Linux Open Sound System (OSS)
- Advanced Linux Sound Architecture (ALSA)
- OpenAL / JOAL

Java AudioClip Interface

- 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…

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

- "Open Audio Library"
  - 3D Audio API (**`www.openal.org`**)
- Open-source
- Cross-platform
- Modeled after OpenGL
- Java binding ("JOAL"):
  `www.jogamp.org`

Playing Sounds in CN1

- **`Media`** object should be created to play sounds.
- **`Media`** objects is created by the overloaded **`creatMedia()`** static method of the **`MediaManager`** class.
- **`createMedia()`** takes in an **`InputStream`** object which is associated to the audio file.
- **`Media, MediaManager`**, and **`InputStream`** are all build-in classes.

Important tips

- You must copy your sound files directly under the **`src`** directory of your project.
- You may need to refresh your project in your IDE (e.g., in Eclipse select the project and hit F5 OR right click on the project and select “Refresh”) for CN1 to properly locate the sound files newly copied to the **`src`** directory.

Creating and playing a sound

```
import java.io.InputStream;
import com.codename1.media.Media;
import com.codename1.media.MediaManager;
...
/** This method constructs a Media object from the
 *  specified file, then plays the Media.
 */
public void playSound (String fileName) {
 try {
   InputStream is = Display.getInstance().getResourceAsStream(getClass(),
                                                              "/"+fileName);
   Media m = MediaManager.createMedia(is, "audio/wav");
   m.play();
   }
 catch (IOException e) {
   e.printStackTrace();
 }
}
//this method calls playSound() to play alarm.wav copied directly under the src directory
public void someOtherMethod(){
   playSound("alarm.wav")
}
```

Encapsulating the sound

```
/** This class encapsulates a sound file as an Media inside a
 *  "Sound" object, and provides a method for playing the Sound.
 */`
public class Sound {
    private Media m;
    public Sound(String fileName) {
        try{
        InputStream is = Display.getInstance().getResourceAsStream(getClass(),
                                                                   "/"+fileName);
        m = MediaManager.createMedia(is, "audio/wav");
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    public void play() {
        //start playing the sound from time zero (beginning of the sound file)
        m.setTime(0);
        m.play();
    }
}
```


Encapsulating the sound (cont)

- In the assignments, you should use encapsulated sounds.
- Create a single sound object for each audio file:

Sound catCollisionSound = new Sound("meow.wav");

Sound scoopSound = new Sound("scoop.wav");

- Operations that belong to the same type should play this single instance (e.g., make all cat-cat collisions call catCollisionSound.play()), instead of creating new instances.

Looping the Sound

- To create a sound which is played in a loop (e.g., the background sound), **`Media`** object **`m`** indicated above should be created differently.
- We must attach a **`Runnable`** object to it which is invoked when the media has finished playing.
- The **`run()`** method of the **`Runnable`** object must play the sound starting from its beginning.

Encapsulating Looping Sound

```
/**This class creates a Media object which loops while playing the sound
 */
public class BGSound implements Runnable{
  private Media m;

  public BGSound(String fileName){
    try{
      InputStream is = Display.getInstance().getResourceAsStream(getClass(),
                                                          "/"+fileName);
      //attach a runnable to run when media has finished playing
      //as the last parameter
      m = MediaManager.createMedia(is, "audio/wav", this);
    }
    catch(Exception e){
      e.printStackTrace();
    }
  }
  public void pause(){ m.pause();} //pause playing the sound
  public void play(){ m.play();} //continue playing from where we have left off

  //entered when media has finished playing
  public void run() {
    //start playing from time zero (beginning of the sound file)
    m.setTime(0);
    m.play();
    }
}
```

Use of Encapsulated Looping Sound

```
/**This form creates a looping sound and a button which pauses/plays the looping sound
 */

public class BGSoundForm extends Form implements ActionListener{

  private BGSound bgSound;
  private boolean bPause = false;
  public BGSoundForm() {
    Button bButton = new Button(“Pause/Play”);
    //...[style and add bButton to the form]
    bButton.addActionListener(this);
    bgSound = new BGSound("alarm.wav");
    bgSound.play();
  }

  public void actionPerformed(ActionEvent evt) {
    bPause = !bPause;
    if (bPause)
      bgSound.pause();
    else
      bgSound.play();
  }
}
`
```

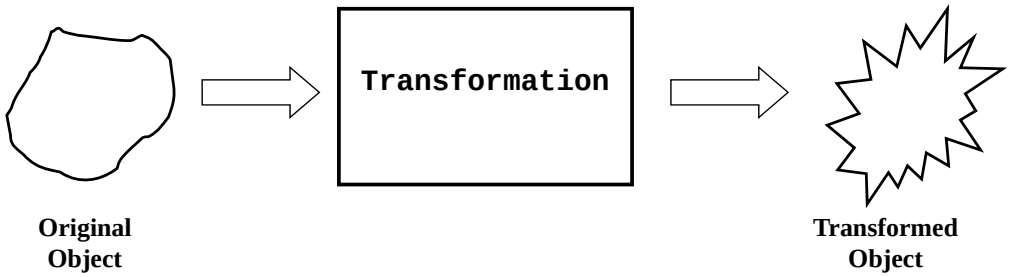
13 - Transformations

Computer Science Department
California State University, Sacramento

Overview

- **Affine Transformations: Translation, Rotation, Scaling**
- **Transforming Points & Lines**
- **Matrix Representation of Transforms**
- **Homogeneous Coordinates**
- **Concatenation of Transformations**

The "Transformation" Concept



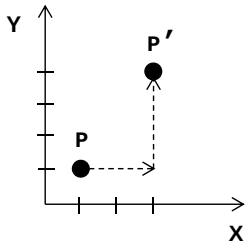
- "Original object" could be anything
  - We will focus on geometric objects
- "Transformed object" is usually (*but not necessarily*) of same type

"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)

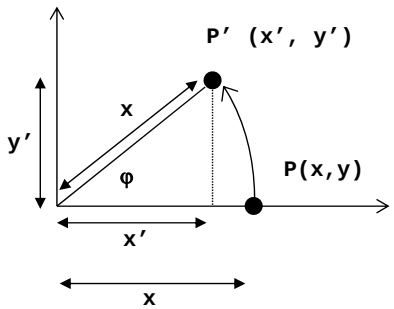
T = (+2, +3)

P' = (x+2, y+3)

P → [T] → P' or P' ← [T] ← P

Transformations on Points (cont.)

- Rotation about the origin (point on X axis)



cos (φ) = x' / x ; hence

x' = x cos(φ)

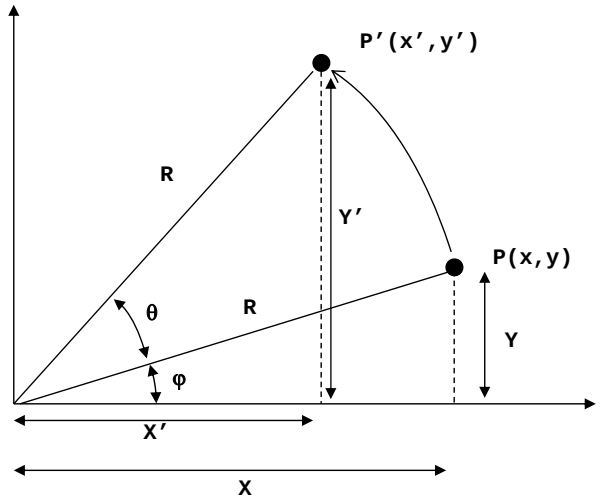
sin (φ) = y' / x ; hence

y' = x sin(φ)

P → [R] → P' or P' ← [R] ← 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 $Y = R\sin(\varphi)$

$X' = R\cos(\varphi + \theta)$

$= R(\cos(\varphi)\cos(\theta) - \sin(\varphi)\sin(\theta))$

$= \underline{R\cos(\varphi)}\cos(\theta) - \underline{R\sin(\varphi)}\sin(\theta)$

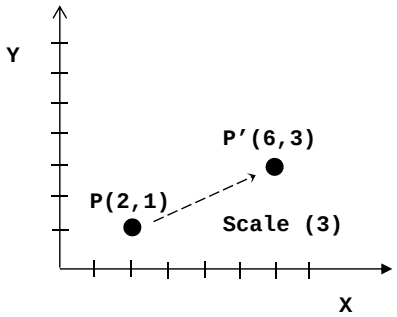
$= \underline{X}\cos(\theta) - \underline{Y}\sin(\theta)$

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.)

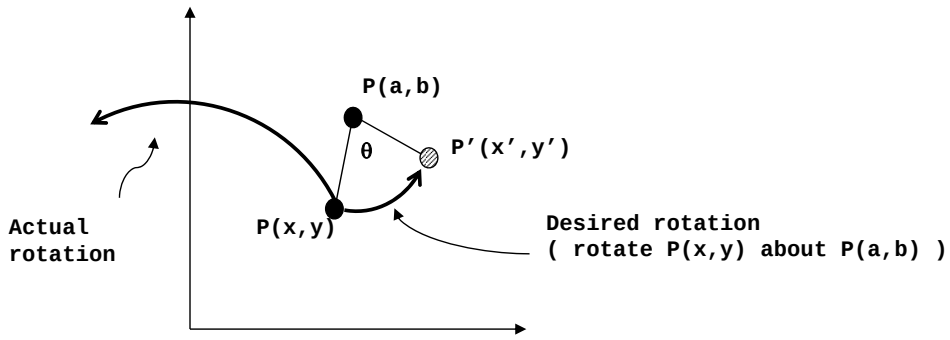
- Scaling is
  - Relative to the origin (like rotation)
  - *Different* from a "move":
    - Translate (3,3) always moves exactly 3 units
    - Scale (3,3) depends on the initial point being scaled:

```
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"

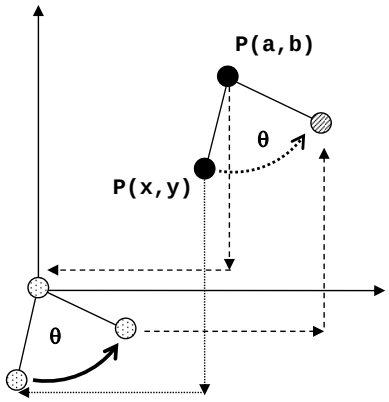
Transformations on Points (cont.)

- Rotating a point about an arbitrary point
  - Problem: rotation formulas are ***relative to the origin***



Transformations on Points (cont.)

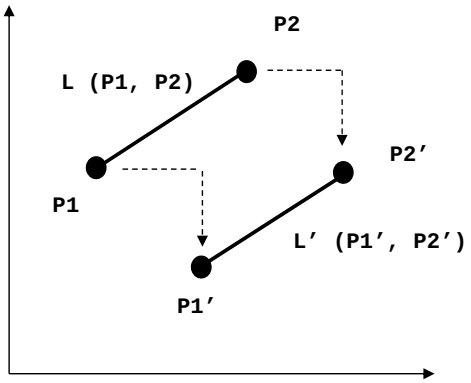
- Solution:
  - Translate to origin
  - Perform rotation
  - Translate "back"



1. **Translate P(x,y) by (-a, -b)**
2. **Rotate (translated) P**
3. **"Undo" the translation (translate result by (+a, +b))**

Transformations on Lines

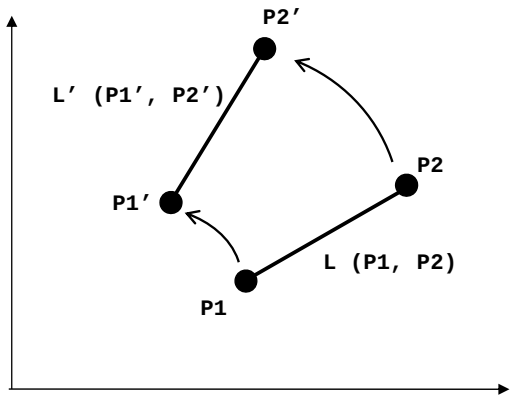
- Translation: translate the endpoints



- **Translate (Line(p1,p2)) = Line (Translate(p1), Translate(p2))**

Transformations on Lines (cont.)

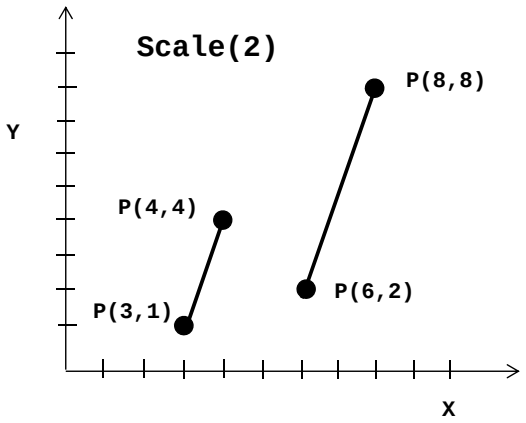
- Rotation *about the origin*: rotate the endpoints



- **Rotate (Line(p1,p2))**
  = Line (Rotate(p1), Rotate(p2))

Transformations on Lines (cont.)

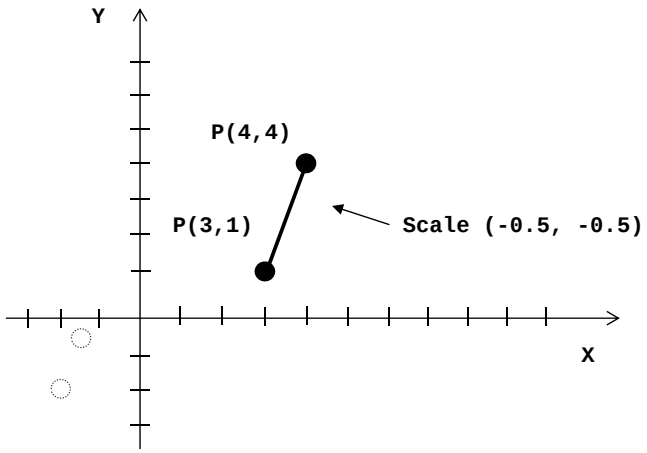
- Scaling: scale the endpoints



- **Scale (Line(p1,p2))**
  = Line (Scale(p1), Scale(p2))
- **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:

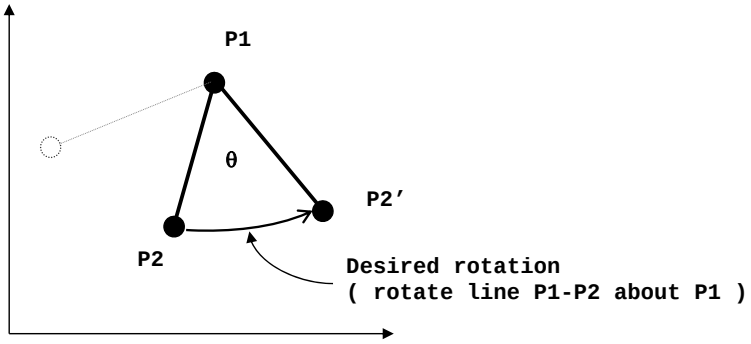
- Absolute Value of Scale Factor > 1 → "bigger"
- Absolute Value of Scale Factor < 1 → "smaller"
- Scale Factor < 0 → "flip" ("mirror")

Identity Operations:

- For translation: 0 → No Change
- For rotation: 0 → No Change
- For scaling: 1 → No Change

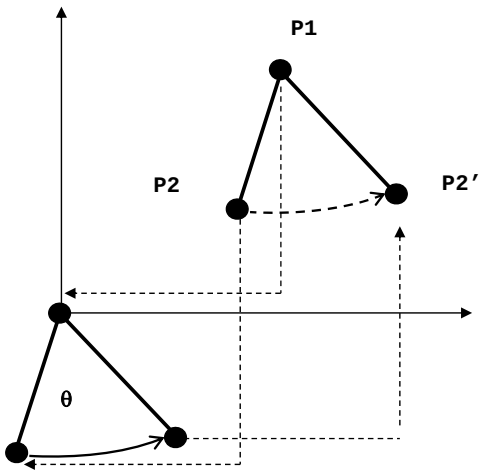
Transformations on Lines (cont.)

- 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 ?



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 = (sx, sy)
P' = (x*sx, y*sy)
```

$$P' = \begin{bmatrix} x & y \end{bmatrix} * \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

$$= \begin{bmatrix} (x * s_x) & (y * s_y) \end{bmatrix}$$

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

$$\begin{bmatrix} x & y & 1 \end{bmatrix} * \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} (x\cos(\theta) - y\sin(\theta)) & (x\sin(\theta) + y\cos(\theta)) & 1 \end{bmatrix}$$

Applying Transformations (cont.)

- Scaling

$$\begin{bmatrix} x & y & 1 \end{bmatrix} * \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} (x * S_x) & (y * S_y) & 1 \end{bmatrix}$$

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:

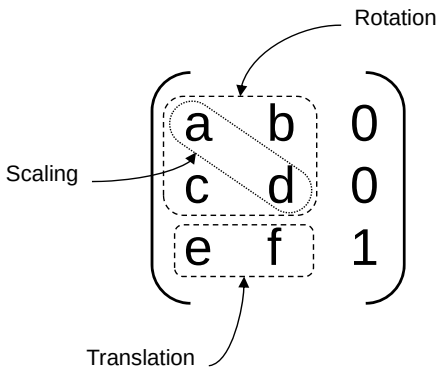
$$\begin{bmatrix} (x\cos(\theta) - y\sin(\theta)) \\ (x\sin(\theta) + y\cos(\theta)) \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Scaling:

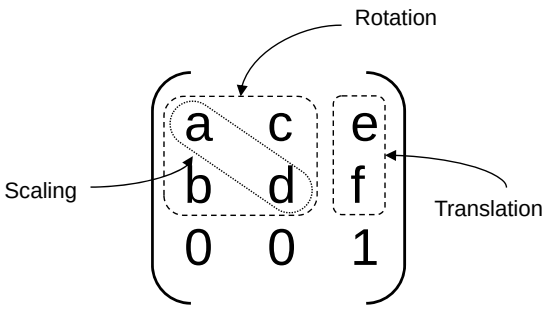
$$\begin{bmatrix} (x * S_x) \\ (y * S_y) \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Active Matrix Areas

Row-major form



Column-major form



Same size “active area” – 6 elements (3x2 or 2x3)

Concatenation of Transforms

Typical Sequence:

$$P1 \times Translate(tx,ty) = P2 ;$$

$$P2 \times Rotate(\theta) = P3 ;$$

$$P3 \times Scale(sx,sy) = P4 ;$$

$$P4 \times Translate(tx,ty) = P5 ;$$

Concatenation of Transforms (cont.)

- In (row-major) Matrix Form:

$$\begin{bmatrix} x1 & y1 & 1 \end{bmatrix} \times \begin{pmatrix} Translate \\ (tx,ty) \end{pmatrix} = \begin{bmatrix} x2 & y2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x2 & y2 & 1 \end{bmatrix} \times \begin{pmatrix} Rotate(\theta) \end{pmatrix} = \begin{bmatrix} x3 & y3 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x3 & y3 & 1 \end{bmatrix} \times \begin{pmatrix} Scale \\ (sx,sy) \end{pmatrix} = \begin{bmatrix} x4 & y4 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x4 & y4 & 1 \end{bmatrix} \times \begin{pmatrix} Translate \\ (tx,ty) \end{pmatrix} = \begin{bmatrix} x5 & y5 & 1 \end{bmatrix}$$

Concatenation of Transforms (cont.)

- Alternate Matrix Form:

$$\left(\left(\left(\left(x1 \;\; y1 \;\; 1\right) \times \left(T1\right)\right) \times \left(R1\right)\right) \times \left(S1\right)\right) \times \left(T2\right)$$

$$= \left(x5 \;\; y5 \;\; 1\right)$$

Concatenation of Transforms (cont.)

- Matrix multiplication is *associative* :

$$\left(x1 \;\; y1 \;\; 1\right) \times \underbrace{\left(\left(T1\right) \times \left(R1\right) \times \left(S1\right) \times \left(T2\right)\right)}_{M} = \left(x5 \;\; y5 \;\; 1\right)$$

$$\left(x1 \;\; y1 \;\; 1\right) \times \left(M\right) = \left(x5 \;\; y5 \;\; 1\right)$$

In Column-Major Form

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} \textbf{\textit{Trans}} \\ \textbf{\textit{(x, y)}} \end{bmatrix} \times \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_3 \\ y_3 \\ 1 \end{bmatrix} = \begin{bmatrix} \textbf{\textit{Rot}} \quad (\theta) \end{bmatrix} \times \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_4 \\ y_4 \\ 1 \end{bmatrix} = \begin{bmatrix} \textbf{\textit{Scale}} \\ \textbf{\textit{(sx, sy)}} \end{bmatrix} \times \begin{bmatrix} x_3 \\ y_3 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_5 \\ y_5 \\ 1 \end{bmatrix} = \begin{bmatrix} \textbf{\textit{Trans}} \\ \textbf{\textit{(x, y)}} \end{bmatrix} \times \begin{bmatrix} x_4 \\ y_4 \\ 1 \end{bmatrix}$$

Column-Major Form (cont.)

$$\begin{bmatrix} x_5 \\ y_5 \\ 1 \end{bmatrix} = \left([T2] \times \left([S1] \times \left([R1] \times \left([T1] \times \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \right) \right) \right) \right)$$

$$\begin{bmatrix} x_5 \\ y_5 \\ 1 \end{bmatrix} = \underbrace{\left([T2] \times [S1] \times [R1] \times [T1] \right)}_{M} \times \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_5 \\ y_5 \\ 1 \end{bmatrix} = [M] \times \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

14 - Applications of Affine Transforms

Computer Science Department
California State University, Sacramento

Overview

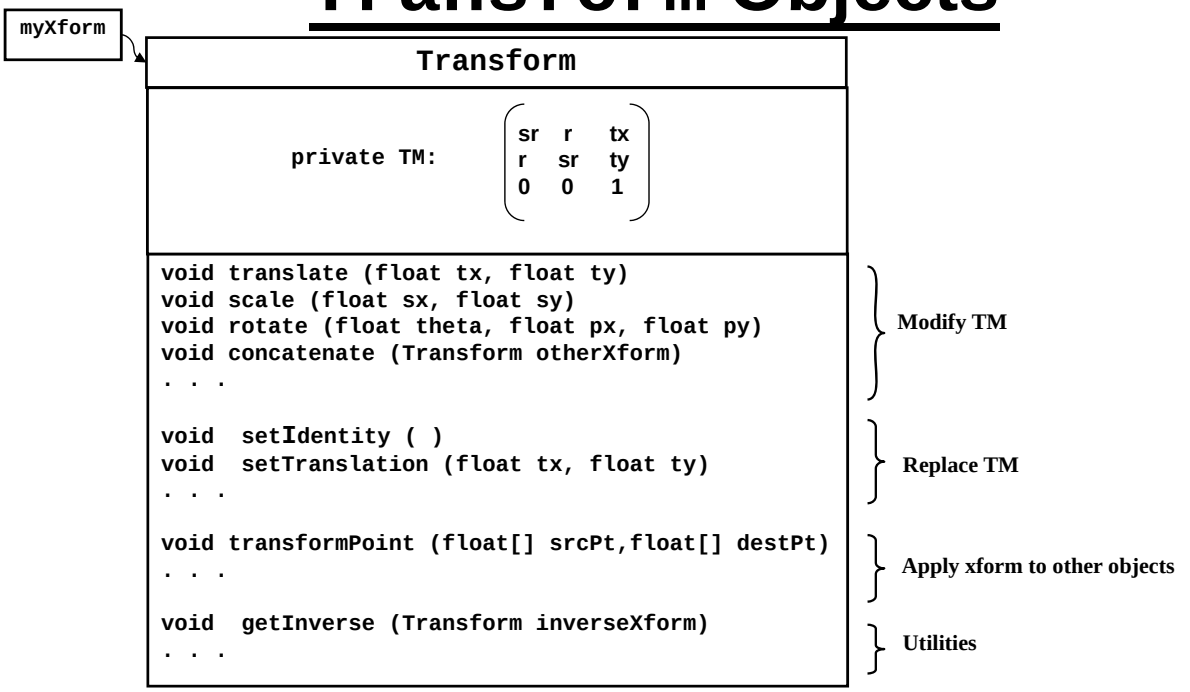
- `Transform` Class
- Local Coordinate Systems
- Display-Mapping Transforms
- `Graphics` Class revisited
- Transformable Objects
- Composite Transforms
- Hierarchical Object Transforms
- Dynamic Transforms

Transform Class

- **com.codename1.ui.Transform**
- **Contains**
  - **A 3×3 "Transformation Matrix" (TM)**
    - **Uses *column-major* form**
    - ***Only the active 2x3 elements can be accessed***
  - **Methods to *manipulate* TM**
  - **Methods to *apply* the transform (xform) to other objects**
- **To initialize use the following static function:**

```
Transform myXform = Transform.makeIdentity();
```


Transform Objects



- Methods for modiying TM (e.g., **`translate(), scale()`** and **`rotate()`**) are always applied relative to the **screen origin** (i.e., coordinates passed to these methods are relative to screen origin).
- Also these methods multiply the new transform to the current TM ***on the right,*** which means the transform concatenated last to the xform will be applied first to a point.

Using `Transform` Object

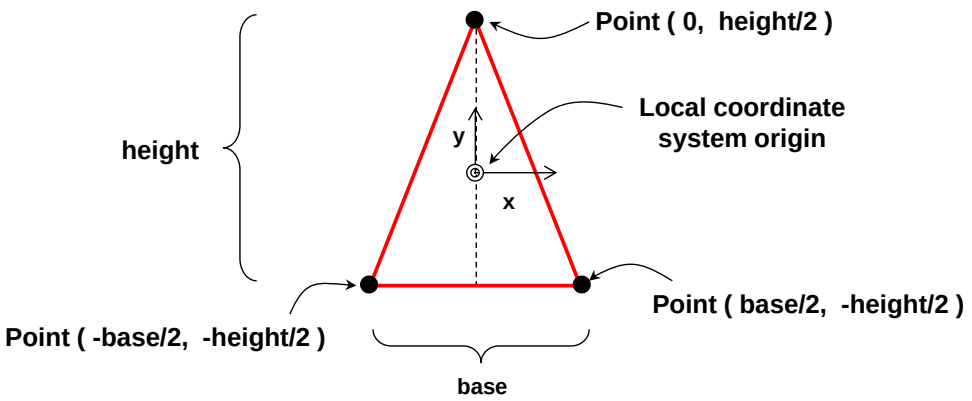
```
...
float[] p1 = new float[]{x,y};
float[] p2 = new float[]{0,0};
Transform myXform = Transform.makeIdentity();
myXform.rotate(Math.toRadians(45), 0, 0);
myXform.transformPoint (p1,p2);
```

$$\begin{bmatrix} x2 \\ y2 \\ 1 \end{bmatrix} = \begin{bmatrix} \quad Rotate(45^0) \quad \end{bmatrix} \times \begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix}$$

"Local" Coordinate Systems

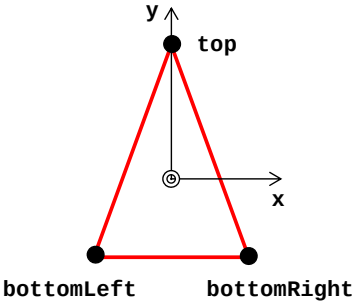
Define objects *relative to their own origin*

- Example: triangle
  - Base & Height
  - Local origin at "center"
  - Points defined *relative to local origin*



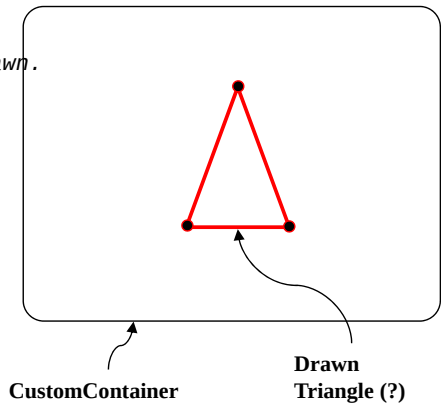
Triangle Class

```
/** This class defines an isosceles triangle with a specified base and height. The triangle
points are defined in "local space", and the local space axis orientation is X to the right
and Y upward. Local origin coincides with the container origin to draw the triangle on the
container. That is why, we pass "triangle point + pCmpRelPrnt" as a drawing coordinate to
the drawLine() method.*/
public class Triangle {
  private Point top, bottomLeft, bottomRight ;
  private int 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 = ColorUtil.BLACK;
}
  public void draw (Graphics g, Point pCmpRelPrnt) {
    g.setColor(color);
    g.drawLine (pCmpRelPrnt.getX()+ top.getX(), pCmpRelPrnt.getY()+top.getY(),
                pCmpRelPrnt.getX()+bottomLeft.getX(),
                pCmpRelPrnt.getY()+bottomLeft.getY());
    g.drawLine (pCmpRelPrnt.getX()+bottomLeft.getX(),
                pCmpRelPrnt.getY()+bottomLeft.getY(),
                pCmpRelPrnt.getX()+bottomRight.getX(),
                pCmpRelPrnt.getY()+bottomRight.getY());
    g.drawLine (pCmpRelPrnt.getX()+bottomRight.getX(),
                pCmpRelPrnt.getY()+bottomRight.getY(),
                pCmpRelPrnt.getX()+top.getX(),
                pCmpRelPrnt.getY()+top.getY());
  }
}
```



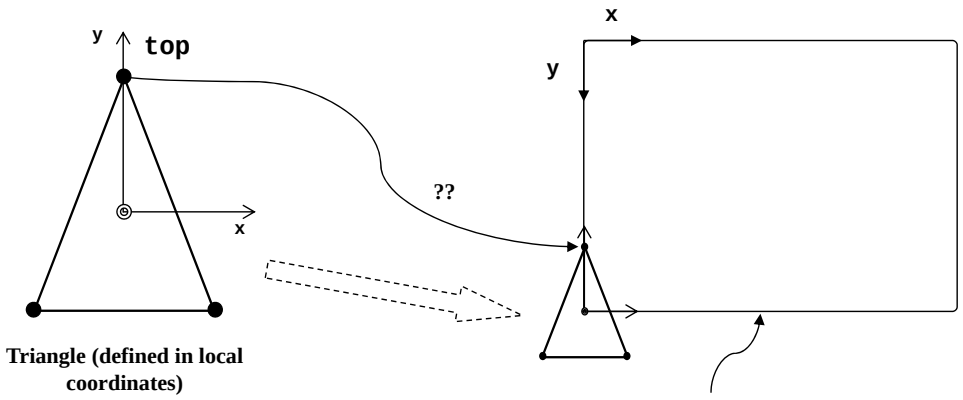
Drawing A Triangle

```
/** This class defines a container that has a triangle.
 *  Repainting the container causes the triangle to be drawn.
 */
public class CustomContainer extends Container{ {
  private Triangle myTriangle ;
  public CustomContainer () {
    myTriangle = new Triangle (200, 200) ;
  }
  public void paint (Graphics g) {
    super.paint (g);
    myTriangle.draw(g, new Point(this.getX(), this.getY())) ;
  }
}
```

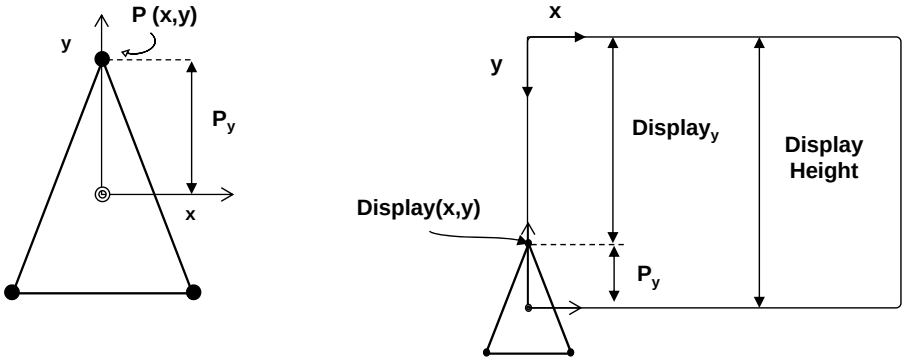


Mapping To Display Location

- Suppose desired location was “centered at lower-left display corner”
- How do we compute location of “top”?



Mapping To Display Location (cont.)



- $Display_x = P_x$
- $Display_y = DisplayHeight - P_y$

$$= \underbrace{(-1*(P_y))}_{Scale_y(-1)} + \underbrace{DisplayHeight}_{Translate_y(DisplayHeight)}$$

Applying the Display Mapping

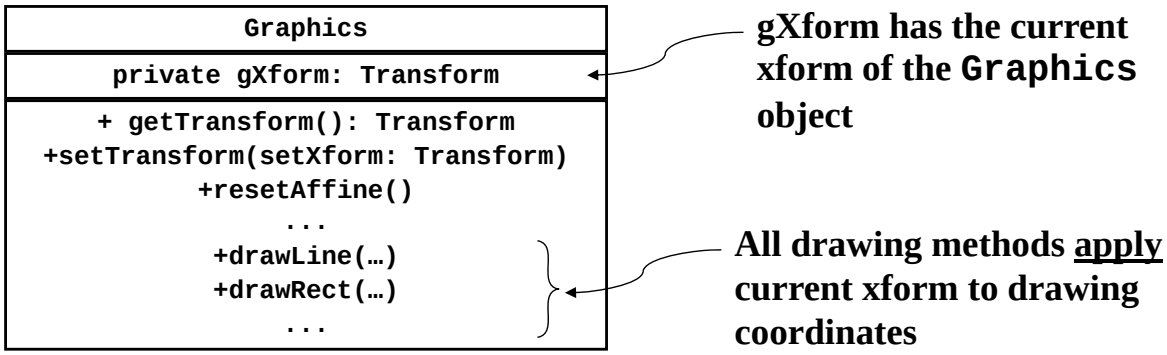
```
/** This class draws an Isosceles Triangle applying "display mapping"
 *  transformations to the triangle's points.
 */
public class Triangle {

  private float[] top, bottomLeft, bottomRight ;
  ...

  public void draw (Graphics g, Point pCmpRelPrnt, int height) {
    // create an displayXform to map triangle points to "display space"
    Transform displayXform = Transform.makeIdentity();
    displayXform.translate (0, height);
    displayXform.scale (1, -1);
    // apply the display mapping transforms to the triangle points
    displayXform.transformPoint(top,top);
    displayXform.transformPoint(bottomLeft,bottomLeft);
    displayXform.transformPoint(bottomRight,bottomRight);

    // draw the (transformed) triangle
    g.setColor(color);
    g.drawLine(pCmpRelPrnt.getX()+(int)top[0], pCmpRelPrnt.getY()+(int)top[1],
      pCmpRelPrnt.getX()+(int)bottomLeft[0],
      pCmpRelPrnt.getY()+(int)bottomLeft[1]); // left side
    g.drawLine(pCmpRelPrnt.getX()+(int)bottomLeft[0],
          pCmpRelPrnt.getY()+(int)bottomLeft[1], pCmpRelPrnt.getX()+
          (int)bottomRight[0], pCmpRelPrnt.getY()+ (int)bottomRight[1]); // bottom
    g.drawLine(pCmpRelPrnt.getX()+(int)bottomRight[0],
          pCmpRelPrnt.getY()+(int)bottomRight[1], pCmpRelPrnt.getX()+(int)top[0],
          pCmpRelPrnt.getY()+(int)top[1]); // right side
  }
}
```


Problems…

- Triangle flips between top and bottom of the display.
- Because the transformations permanently alter the triangle points.
- We could solve this by using *temporary variables* for the transformed points.
- There is a better solution which does not require us to transform the triangle points (this solution will allow us to directly use the points that are defined relative to the local origin).

The Graphics Class

- Every **Graphics** contains a **Transform** object
  - This **transform is applied to all drawing coordinates** during drawing



Using Graphics's Xform

- We can concatenate scale and translate associated with the display mapping to the current xform of the **Graphics** object. Then tell the triangle to draw itself using that **Graphics** object.
- This causes the specified scale and translate to be applied to the drawing coordinates when the triangle is drawn.
- To draw the triangle on display (**CustomContainer**), the local origin coincides with the display origin.
- Remember that this origin is positioned at **(getX(), getY())** relative to component's parent container origin (origin of the content pane of the form) and point **pCmpRelPrnt** contains this position.
- That is why, a drawing coordinate is positioned at "triangle point + **pCmpRelPrnt**" relative to parent origin and we pass this value to the **drawLine()** method which expects coordinates relative to parent origin.

Using Graphics's Xform (contd.)

- However, since transformations are applied relative to the screen origin (i.e., coordinates passed to transformation methods are relative to screen origin), we first need to move the drawing coordinates so that local origin coincides with the screen origin.
- Remember that local origin (positioned at **`(getX(), getY())`** relative to component's parent container origin) is positioned at **`(getAbsoluteX(), getAbsoluteY())`** relative to the screen origin.
- Hence a drawing coordinate positioned at "triangle point + **`pCmpRelPrnt`**" relative to parent origin is located at "triangle point + **`pCmpRelScrn`**" relative to screen origin where points **`pCmpRelPrnt and pCmpRelScrn`** contains **`(getX(), getY())`** and **`(getAbsoluteX(), getAbsoluteY())`** values, respectively.
- That is why, before we apply scale and translate associated with display mapping, we need to move the drawing coordinates by **`translate(-getAbsoluteX(), -getAbsoluteY())`** (**`translate()`**, like other transformation methods, expects us to provide coordinates relative to the screen origin).

Using Graphics's Xform (contd.)

- After applying display mapping we need to move the drawing coordinates back to where they were by **`translate(getAbsoluteX(), getAbsoluteY())`** so that we can draw the triangle on the display (CustomContainer).
- We call these translations related with moving the drawing coordinates back and forth (so that local origin coincides with screen origin before the display mapping is done) as "**local origin**" transformation.
- After triangle is drawn, we need to restore the original xform (the xform before the display mapping and local origin transformations are applied) of the **`Graphics`** object since graphics object is used for other operations after the **`paint()`** returns. **`resetAffine()`** method of **`Graphics`** class is used for this purpose.

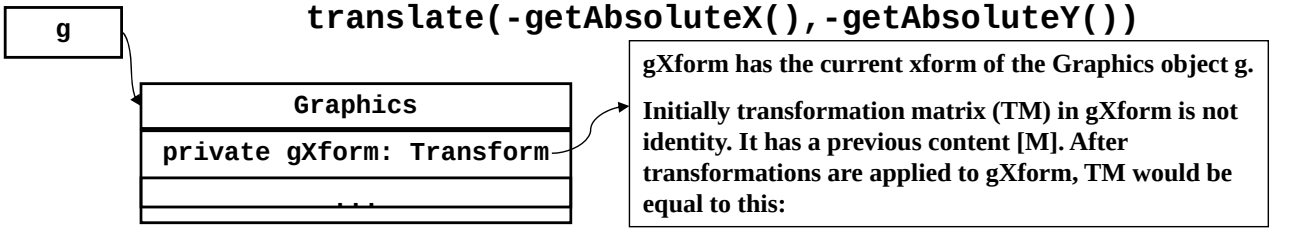
Using Graphics's Xform (cont.)

```
public class CustomContainer extends Container {
  private Triangle myTriangle ;
  public CustomContainer () {
    myTriangle = new Triangle (200, 200);
  }
  public void paint (Graphics g) {
    super.paint(g);
    Transform gXform = Transform.makeIdenitity();
    g.getTransform(gXfrom);
    //move drawing coordinates back
    gXform.translate(getAbsoluteX(),getAbsoluteY());
    //apply translate associated with display mapping
    gXform.translate(0, getHeight());
    //apply scale associated with display mapping
    gXform.scale(1, -1);
    //move drawing coordinates so that the local origin coincides with the screen origin
    gXform.translate(-getAbsoluteX(),-getAbsoluteY());
    g.setTransform(gXform);
    myTriangle.draw(g, new Point(getX(), getY()));
    //restore the original xform in g
    g.resetAffine();
  }
}
```


Using Graphics's Xform (cont.)

- Effect of modifying **g**'s transform in **paint()** :

```
translate(getAbsoluteX(),getAbsoluteY())
translate(0, getHeight());
scale(1,-1);
translate(-getAbsoluteX(),-getAbsoluteY())
```



$$[M] \times [T(absX, absY)] \times [T_y(displayHeight)] \times [S_y(-1)] \times [T(-absX, -absY)]$$

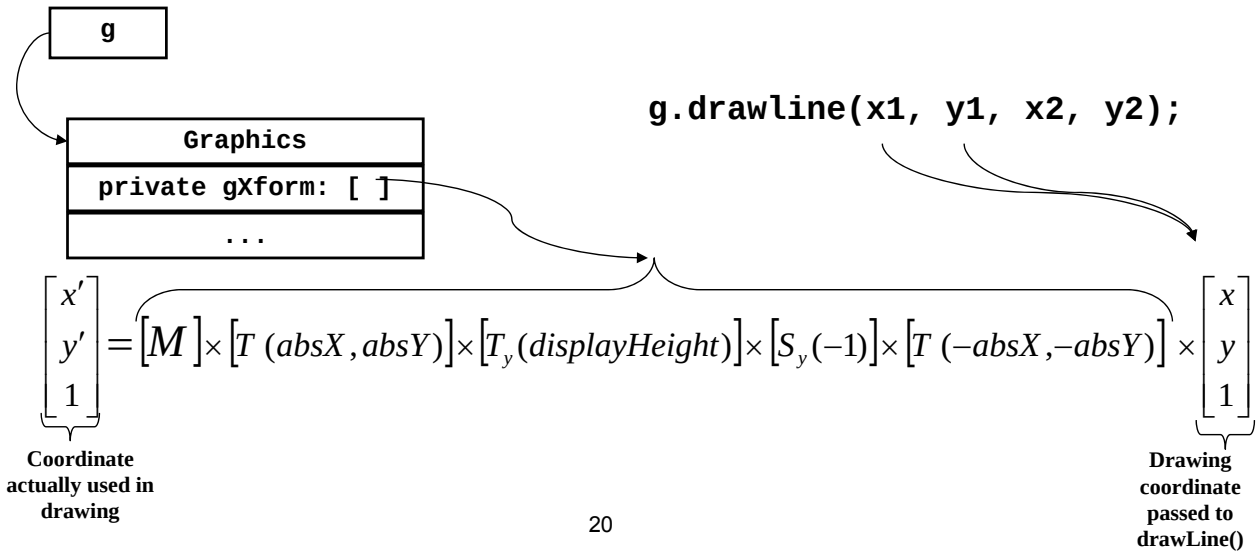
Previous content

Using Graphics's Xform (cont.)

```
/** This class defines a triangle, as before.
 *  The Graphics object applies its current xform to all drawing
 *  coordinates prior to performing any output operation.
 */
public class Triangle {
  private Point top, bottomLeft, bottomRight ;
  private int 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 = ColorUtil.BLACK;
  }
  public void draw (Graphics g, Point pCmpRelPrnt) {
    g.setColor(color);
    g.drawLine (pCmpRelPrnt.getX()+ top.getX(), pCmpRelPrnt.getY()+top.getY(),
                pCmpRelPrnt.getX()+bottomLeft.getX(),
                pCmpRelPrnt.getY()+bottomLeft.getY());
    g.drawLine (pCmpRelPrnt.getX()+bottomLeft.getX(),
                pCmpRelPrnt.getY()+bottomLeft.getY(),
                pCmpRelPrnt.getX()+bottomRight.getX(),
                pCmpRelPrnt.getY()+bottomRight.getY());
    g.drawLine (pCmpRelPrnt.getX()+bottomRight.getX(),
                pCmpRelPrnt.getY()+bottomRight.getY(),
                pCmpRelPrnt.getX()+top.getX(),
                pCmpRelPrnt.getY()+top.getY());
  }
}
```

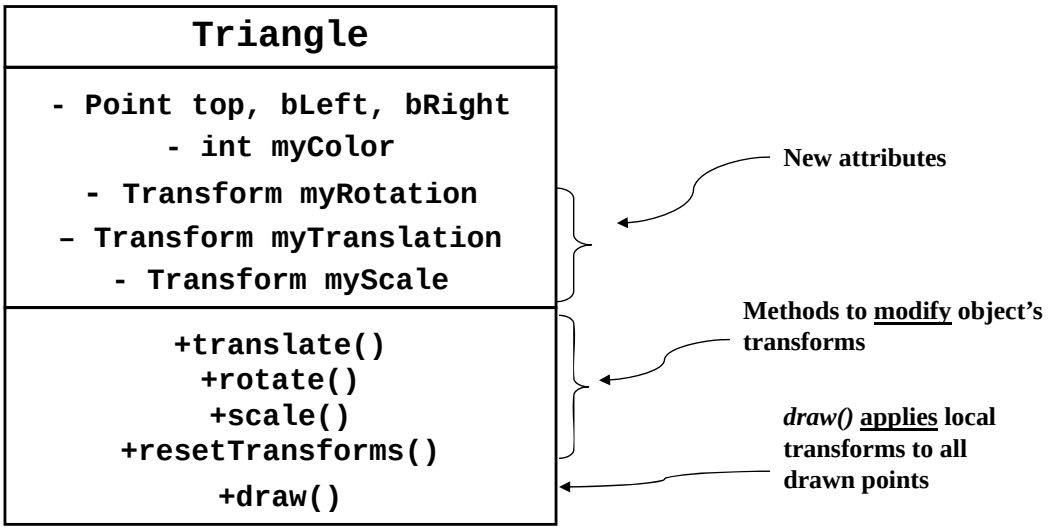

Using Graphics's Xform (cont.)

- Effect of using **g** to draw a line in **Triangle.draw()**:



Transformable Objects

- Expand objects to contain *"local transforms" (LTs)*
- Arrange to *apply an object's transforms* when it is drawn




```
/** This class defines a triangle with Local Transformations (LTs). Client
 *  code can apply arbitrary transformations to the triangle by invoking methods to
 *  update/modify the LTs; when the triangle is drawn it automatically
 *  applies its current LTs to drawing coordinates. */
public class Triangle {
  private Point top, bottomLeft, bottomRight ;
  private int myColor ;
  private Transform myRotation, myTranslation, myScale ;
  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 = ColorUtil.BLACK ;
    myRotation = Transform.makeIdentity();
    myTranslation = Transform.makeIdentity();
    myScale = Transform.makeIdentity();
  }
  public void rotate (float degrees) {
    //pivot point (rotation origin) is (0,0), this means the rotation will be applied about
    //the screen origin
    myRotation.rotate ((float)Math.toRadians(degrees),0,0);
  }
  public void translate (float tx, float ty) {
    myTranslation.translate (tx, ty);
  }
  public void scale (float sx, float sy) {
    //remember that like other transformation methods, scale() is also applied relative to
    //screen origin
    myScale.scale (sx, sy);
  }
  //...continued...
```

Transformable Objects (cont.)

```
// ... Triangle class, cont.
public void resetTransform() {
 myRotation.setToIdentity();
 myTranslation.setToIdentity();
 myScale.setToIdentity();
 }
/* This method applies the triangle's LTs to the received Graphics object's xform, then uses this
xform (with the additional transformations) to draw the triangle. Note that we pass getAbsoluteX()
and getAbsoluteY() values of the container as pCmpRelScrn*/
 public void draw (Graphics g, Point pCmpRelPrnt, Point pCmpRelScrn) {
  // set the drawing color for the triangle
  g.setColor(myColor);
  //append the triangle's LTs to the xform in the Graphics object. But first move the drawing
  //coordinates so that the local origin coincides with the screen origin. After LTs are applied,
  //move the drawing coordinates back.
  Transform gXform = Transform.makeIdenity();
  g.getTransform(gXform);
  gXform.translate(pCmpRelScrn.getX(),pCmpRelScrn.getY());
  gXform.translate(myTranslation.getTranslateX(), myTranslation.getTranslateY());
  gXform.concatenate(myRotation);
  gXform.scale(myScale.getScaleX(), myScale.getScaleY());
  gXform.translate(-pCmpRelScrn.getX(),-pCmpRelScrn.getY());
  g.setTransform(gXform);
  //draw the lines as before
  g.drawLine(pCmpRelPrnt.getX()+top.getX(), pCmpRelPrnt.getY()+top.getY(),
    pCmpRelPrnt.getX() + bottomLeft.getX(),pCmpRelPrnt.getY() + bottomLeft.getY());
  //...[draw the rest of the lines]
  }} //end of Triangle class
```



```
/** This class defines a container 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 container's paint() method applies the "display mapping"
transformation to the Graphics object, and tells the triangle to "draw itself".  The triangle
applies its LTs to the Graphics object in its draw() method.
 */
public class CustomContainer extends Container {
 private Triangle myTriangle ;
 public CustomContainer () {
  myTriangle = new Triangle (200, 200) ;      //construct a Triangle
  //apply some transformations to the triangle
  myTriangle.translate (300, 300);
  myTriangle.rotate (90);
  myTriangle.scale (2, 1);
  }
 public void paint (Graphics g) {
  super.paint (g);
  //...[apply the "Display mapping" transformation to the Graphics object as before. But,
  //again as before, first move the drawing coordinates so that the local origin coincides with
  //the screen origin. After display mapping is applied, move the drawing coordinates back.]

  //origin location of the component (CustomContainer) relative to its parent container origin
  Point pCmpRelPrnt = new Point(getX(),getY());
  //origin location of the component (CustomContainer) relative to the screen origin
  Point pCmpRelScreen = new Point(getAbsoluteX(),getAbsoluteY());
  //tell the triangle to draw itself
  myTriangle.draw(g, pCmpRelPrnt, pCmpRelScreen);
  }
}
```

Composite Transforms

- Transformations applied to triangle's drawing coordinates:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} M \end{bmatrix} \times \begin{bmatrix} T_{display} \end{bmatrix} \times \begin{bmatrix} S_{display} \end{bmatrix} \times \begin{bmatrix} T_{tri} \end{bmatrix} \times \begin{bmatrix} R_{tri} \end{bmatrix} \times \begin{bmatrix} S_{tri} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Coordinate actually used in drawing — **Original gXform content** — **Display mapping transformations** — **Triangle's local transformations** — **drawing coordinate passed to drawLine()**

Order of application of transformations

Also called called the "Graphics Transform Stack"

Note: there are also translations applied before and after "display mapping" and "local" transformations which belong to the "local origin" transformations. For brevity, they are not indicated in the above formula.

On Transform Order and Number of LTs

- Suppose an interactive program implements:

  Click = translate (10,10), Drag = rotate (45°)

- "Suppose" the expected result for the interactive sequence "$Drag_1$, $Click_1$, $Drag_2$, $Click_2$" is:
  - Rotation by a total of 90°, Translation by a total of (20,20)

(One might instead want the transformations applied "in sequence", but suppose that is not what we want here…)

- If we only have one LT object, after the above interaction it would look like:

$$[I]\times[R_1(45)]\times[T_1(10,10)]\times[R_2(45)]\times[T_2(10,10)]$$

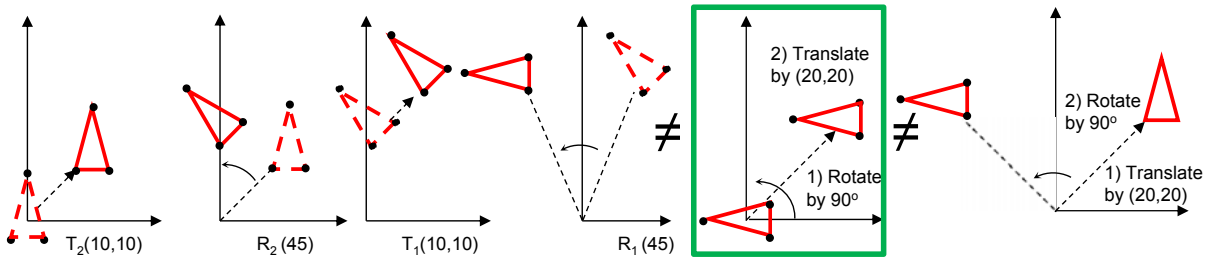
(by default xform is an identity matrix and it is modified by multiplications ***on the right***)

On Transform Order and Number of LTs (cont.)

- When LT is applied to the points defined in the local coordinates, it has the following effect:

$$[I]\times[R_1(45)]\times[T_1(10,10)]\times[R_2(45)]\times[T_2(10,10)]\times\begin{bmatrix}X\\Y\\1\end{bmatrix}$$

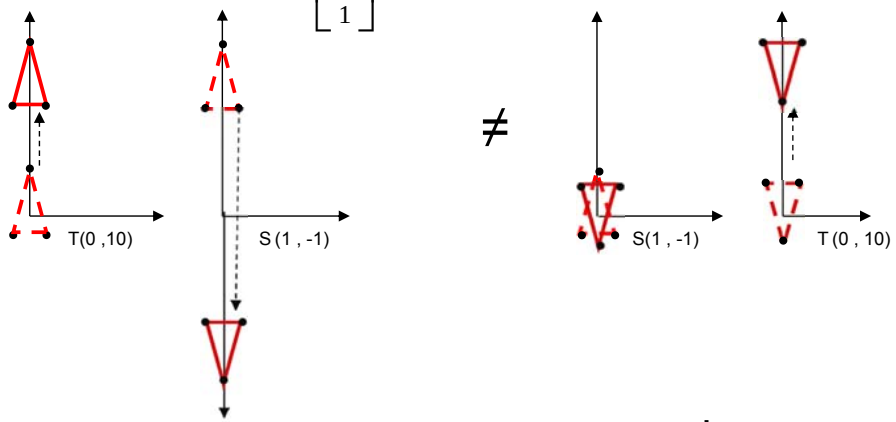
(multiply ***from right to left***: last transform is applied first)



- So to get the expected result we need to accumulate translations and rotations in two separate LTs and rotate the points before translating them (just like the above mentioned Triangle class).
- When we apply scale (e.g., before or after translation) is also equally important…

On Transform Order and Number of LTs (cont.)

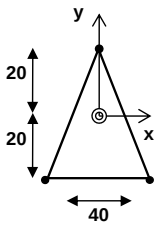
$$[I]\times[S(1,-1)]\times[T(0,10)]\times\begin{bmatrix}X\\Y\\1\end{bmatrix} \neq [I]\times[T(0,10)]\times[S(1,-1)]\times\begin{bmatrix}X\\Y\\1\end{bmatrix}$$



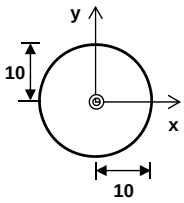
If Click = translate (0,10), Drag = scale (1, 2) and the expected result for "$Drag_1$, $Click_1$, $Drag_2$, $Click_2$" is: "Scaling the height of triangle by x4, and Translation by a total of (0,20)"
Then we should have a separate transform for accumulating scaling transformations too…(Then we would use these separate LTs, in a way that the points would be scaled before they are translated. If we use a single LT, the height would still be scaled by x4, but the triangle would be translated more than 20 units along the Y axis.)

Hierarchical Objects

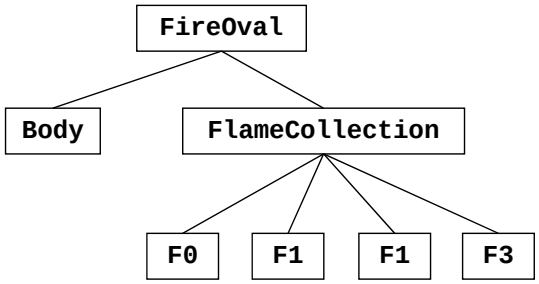
- We can build an object by combining
  - Simpler “parts”
  - Transformations to “orient” the parts



A “Flame” object



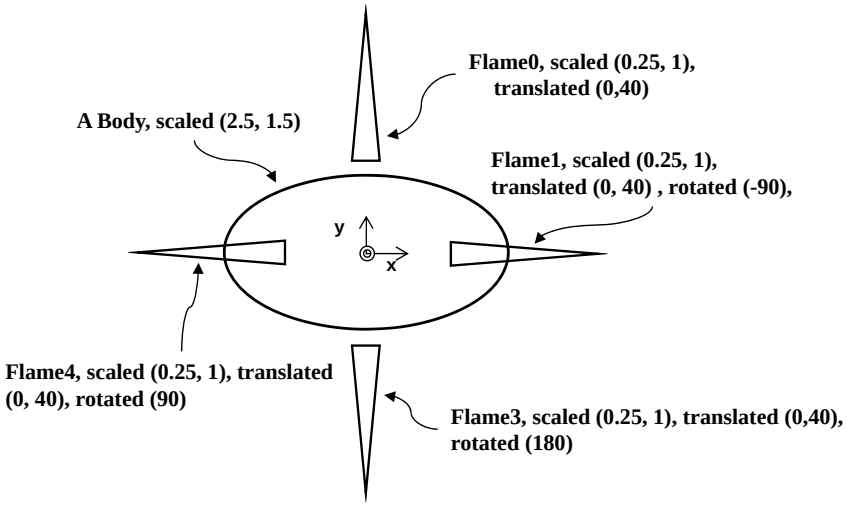
A “Body” object



A hierarchical “FireOval” object

Hierarchical Objects (cont.)

- FireOval Transformations



A hierarchical “FireOval” object

Then we scale the FireOval object with (2, 2) and rotate with 45 degrees and translate it by (400, 200) and apply “display mapping” and “local origin” transformations to it!

Hierarchical Objects (cont.)

```
/** Defines a single "flame" to be used as an arm of a FireOval.
 *  The Flame is modeled after the "Triangle" class, but specifies
 *  fixed dimensions of 40 (base) by 40 (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 int myColor ;
  private Transform myTranslation ;
  private Transform myRotation ;
  private Transform myScale ;

  public Flame (){
    // define a default flame with base=40, height=40, and origin in the center.
    top = new Point (0, 20);
    bottomLeft = new Point (-20, -20);
    bottomRight = new Point (20, -20);

    // initialize the transformations applied to the Flame
    myTranslation = Transform.makeIdentity();
    myRotation = Transform.makeIdentity();
    myScale = Transform.makeIdentity();
  }
  public void setColor(int iColor){
    myColor = iColor;
  }
  //...continued
```



```
// Flame class, continued...
public void rotate (double degrees)        {
  myRotation.rotate (Math.toRadians(degrees), 0, 0);}
public void scale (double sx, double sy) {
  myScale.scale (sx, sy);}
public void translate (double tx, double ty) {
  myTranslation.translate (tx, ty);}

public void draw (Graphics g, Point pCmpRelPrnt, Point pCmpRelScrn) {
  //append the flames's LTs to the xform in the Graphics object (do not forget to do "local
  //origin" transformations). ORDER of LTs: Scaling LT will be applied to coordinates FIRST,
  //then Translation LT, and lastly Rotation LT. Also restore the xform at the end of draw() to
  //remove this sub-shape's LTs from xform of the Graphics object. Otherwise, we would also
  //apply these LTs to the next sub-shape since it also uses the same Graphics object.
  Transform gXform = Transform.makeIdentitiy()
  g.getTransform(gXform);
  Transform gOrigXform = gXform.copy(); //save the original xform
  gXform.translate(pCmpRelScrn.getX(),pCmpRelScrn.getY());
  gXform.concatenate(myRotation); ← Rotation is LAST
  gXform.translate(myTranslation.getTranslateX(), myTranslation.getTranslateY());
  gXform.scale(myScale.getScaleX(), myScale.getScaleY());
  gXform.translate(-pCmpRelScrn.getX(),-pCmpRelScrn.getY());
  g.setTransform(gXform);
  //draw the lines as before
  g.drawLine(pCmpRelPrnt.getX()+top.getX(), pCmpRelPrnt.getY()+top.getY(),
    pCmpRelPrnt.getX() + bottomLeft.getX(),pCmpRelPrnt.getY() + bottomLeft.getY());
  //...[draw the rest of the lines]
  g.setTransform(gOrigXform); //restore the original xform (remove LTs)
  //do not use resetAffine() in draw()! Instead use getTransform()/setTransform(gOrigForm)
  }
} // end of Flame class
```

```
/** Defines a "Body" for a FireOval; the "body" is just a scalable circle with its origin in the
center. Lower left corner in local space would correspond to upper left corner on screen */
public class Body {
  private int myRadius, myColor ;
  private Transform myTranslation, myRotation, myScale ;
 public Body () {
   myRadius = 10;
   Point lowerLeftInLocalSpace = new Point(-myRadius, -myRadius);
   myColor = Color.yellow ;
   myTranslation = Transform.makeIdentity();
   myRotation = Transform.makeIdentity();
   myScale = Transform.makeIdentity(); }
  // ...[code here implementing rotate(), scale(), and translate() as in the Flame class]
 public void draw (Graphics g , Point pCmpRelPrnt, Point pCmpRelScrn) {
  g.setColor(myColor);
  Transform gXform = Transform.makeIdentity();
  g.getTransform(gXform);
  Transform gOrigXform = gXform.copy(); //save the original xform
  gXform.translate(pCmpRelScrn.getX(),pCmpRelScrn.getY());
  gXform.translate(myTranslation.getTranslateX(), myTranslation.getTranslateY());
  gXform.concatenate(myRotation); ← Rotation is not LAST
  gXform.scale(myScale.getScaleX(), myScale.getScaleY());
  gXform.translate(-pCmpRelScrn.getX(),-pCmpRelScrn.getY());
  g.setTransform(gXform);
  //draw the body
  g.fillArc( pCmpRelPrnt.getX() + lowerLeftInLocalSpace.getX(),
             pCmpRelPrnt.getY() + lowerLeftInLocalSpace.getY(),
             2*myRadius, 2*myRadius, 0, 360);
  g.setTransform(gOrigXform); //restore the original xform
  }}
```



```
/** This class defines a "FireOval", which is a hierarchical object composed
 *  of a scaled "Body" and four scaled, translated, and rotated "Flames".
 */
public class FireOval {
  private Body myBody ;
  private Flame [] flames ;
  private Transform myTranslation, myRotation, myScale ;
  public FireOval () {
    myTranslation = Transform.makeIdentity();
    myRotation = Transform.makeIdentity();
    myScale = Transform.makeIdentity();
    myBody = new Body();          // create a properly-scaled Body for the FireOval
    myBody.scale(2.5, 1.5);
    flames = new Flame [4];       // create an array to hold the four flames
    // create four flames, each scaled, translated "up" in Y, and then rotated
    // relative to the local origin
    Flame f0 = new Flame();  f0.translate(0, 40);   f0.scale (0.25, 1);
    flames[0] = f0 ;   f0.setColor(ColorUtil.BLACK);
    Flame f1 = new Flame(); f1.translate(0, 40);f1.rotate(-90);f1.scale(0.25, 1);
    flames[1] = f1 ;   f1.setColor(ColorUtil.GREEN);
    Flame f2 = new Flame(); f2.translate(0, 40);f2.rotate(180);f2.scale(0.25, 1);
    flames[2] = f2 ;   f2.setColor(ColorUtil.BLUE);
    Flame f3 = new Flame(); f3.translate(0, 40);f3.rotate(90);f3.scale(0.25, 1);
    flames[3] = f3;    f3.setColor(ColorUtil.MAGENTA);
  }
  // continued...
```

Hierarchical Objects (cont.)

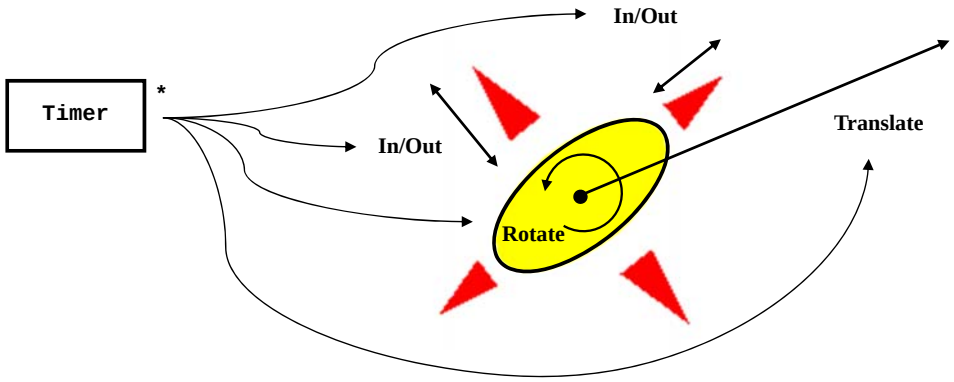
```
   // FirebOval class, continued...
   // ...[code here implementing rotate(), scale(), and translate() as in the Flame class]
 public void draw (Graphics g) {
   Transform gXform = Transform.makeIdentity();
   g.getTransform(gXform);
   Transform gOrigXform = gXform.copy(); //save the original xform
   //move the drawing coordinates back
   gXform.translate(pCmpRelScrn.getX(),pCmpRelScrn.getY());
   // append FireOval's LTs to the graphics object's transform
   gXform.translate(myTranslation.getTranslateX(), myTranslation.getTranslateY());
   gXform.concatenate(myRotation);
   gXform.scale(myScale.getScaleX(), myScale.getScaleY());
   //move the drawing coordinates so that the local origin coincides with the screen origin
   gXform.translate(-pCmpRelScrn.getX(),-pCmpRelScrn.getY());
   g.setTransform(gXform);
   //draw sub-shapes of FireOval
   myBody.draw(g, pCmpRelPrnt, pCmpRelScrn);
   for (Flame f : flames) {
       f.draw(g, pCmpRelPrnt, pCmpRelScrn);
     }
   g.setTransform(gOrigXform);//restore the original xform
   }
 } //end of FireOval class
```



```
/** This class displays a "FireOval" object, scaling, rotating, and translating it into position
on the screen, and telling it to draw itself. Note that CustomContainer object is created by a
form. Code for the form is not provided. It basically sets up GUI using border layout, adds
buttons to north, south, and west containers, and CustomContainer object to center.*/
public class CustomContainer extends Container {
  FireOval myFireOval ;
  public CustomContainer () {
    // create a FireOval to display
    myFireOval = new FireOval ();
    // rotate, scale, and translate this FireOval on the container
    myFireOval.scale(2,2);
    myFireOval.rotate (45) ;
    myFireOval.translate (400, 200) ;      }
  public void paint (Graphics g) {
    super.paint (g);
    Transform gXform = Transform.makeTransform();
    g.getTransform(gXform);
    //move the drawing coordinates back
    gXform.translate(getAbsoluteX(),getAbsoluteY());
    //apply display mapping
    gXform.translate(0, getHeight());
    gXform.scale(1, -1);
    //move the drawing coordinates as part of the "local origin" transformations
    gXform.translate(-getAbsoluteX(),-getAbsoluteY());
    g.setTransform(gXform);
   Point pCmpRelPrnt = new Point(this.getX(), this.getY());
    Point pCmpRelScrn = new Point(getAbsoluteX(),getAbsoluteY());
    // tell the fireball to draw itself
    myFireOval.draw(g, pCmpRelPrnt, pCmpRelScrn);
    g.resetAffine(); //restore the xform in Graphics object
  } } //do not use getTransform()/setTranform(gOrigXform) in paint()!
      //instead use resetAffine()
```

Dynamic Transformations

- We can alter an object's transforms "on-the-fly"
  - Vary sub-shapes (i.e., body and flames) local transforms
  - Vary entire object (i.e., FireOval) local transforms



Dynamic Transformations (cont.)

```
/** This class defines a Form containing a CustomContainer object that displays
 * the FireOval. It uses a Timer to call updateLTs() which modify FireOval's and
 * its Flames' local transformations.
 * CustomContainer class looks exactly like the one used in static FireOval
 * example expect it also has a getFireOval() method that returns FireOval object.
 */
public class DynamicFireOvalForm extends Form implements Runnable {
   private CustomContainer myCustomContainer = new CustomContainer();

   public DynamicFireOvalForm () {
      //...[set up GUI using border layout, add buttons to north, south, and
      //west containers, and CustomContainer object to the center container.]
      UITimer timer = new UITimer(this);
      timer.schedule(10, true, this);
   }
   public void run () {
      myCustomContainer.getFireOval().updateLTs() ;
      myCustomContainer.repaint() ;
   }
```

Dynamic Transformations (cont.)

```
/** This class defines a FireOval object which supports dynamic alteration
 *  of both the FireOval position & orientation, and also of the offset of
 *  the flames from the body.
 */
public class FireOval {
   //...declarations here for Body, Flames, and FireOval transforms, as before;
   // and code here to define the FireOval body and flames, and to define
   // methods for applying transformations, as before...draw() method is as before too…
   private double flameOffset = 0 ;        // current flame distance from FireOval
   private double flameIncrement = 1 ;     // change in flame distance each tick
   private double maxFlameOffset = 10 ;    // max distance before reversing

   // Invoked to update the local transforms of FireOval and its sub-shapes, flames.
   public void updateLTs () {
   // update the FireOval position and orientation
   this.translate(1,1);
   this.rotate(1) ;
   // update the flame positions (move them along their local Y axis)
   // this is why flames are TRANSLATED before they are ROTATED
   for (Flame f:flames) {
      f.translate ((float)0, (float)flameIncrement);
   }
   flameOffset += flameIncrement ;
   // reverse direction of flame movement for next time if we've hit the max
   if (Math.abs(flameOffset) >= maxFlameOffset) {
      flameIncrement *= -1 ;
   } }
```

15 - Viewing Transformations

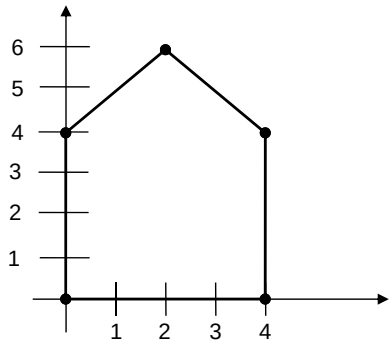
Computer Science Department
California State University, Sacramento

Overview

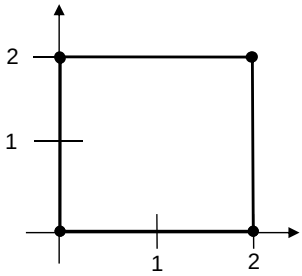
- **The World Coordinate System**
- **Mapping From World to Display Coordinates**
  - **World Window, Normalized Device (ND), World-to-ND Transform, ND-to-Display Transform, the Viewing Transformation Matrix (VTM)**
- **2D Viewing Operations (Zoom and Pan)**
- **Mapping User Input to World Coordinates**
- **Clipping and the Cohen-Sutherland Algorithm**

Local Coordinate Systems

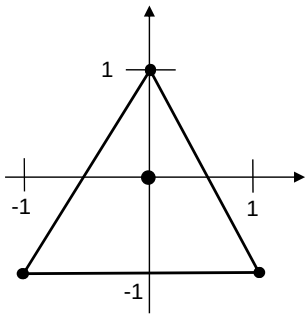
- Each object is defined in its “own space”



Pentagon

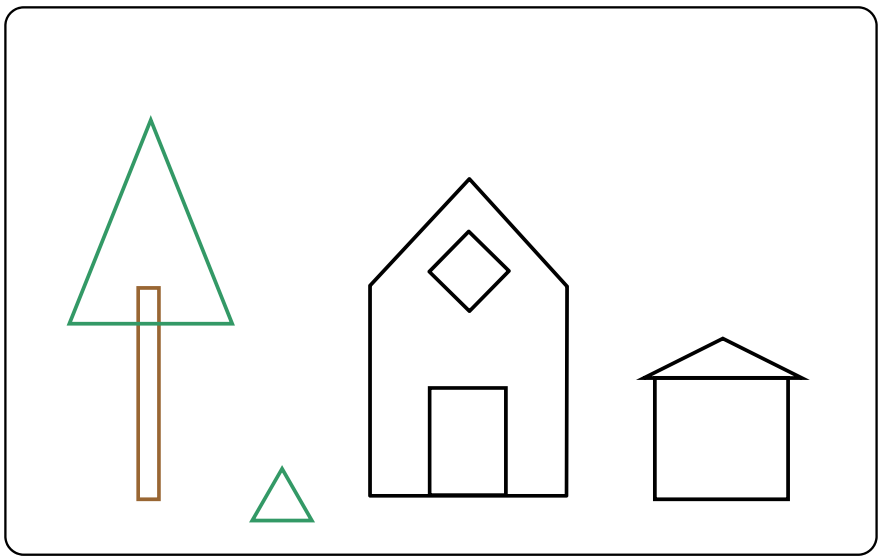


Box



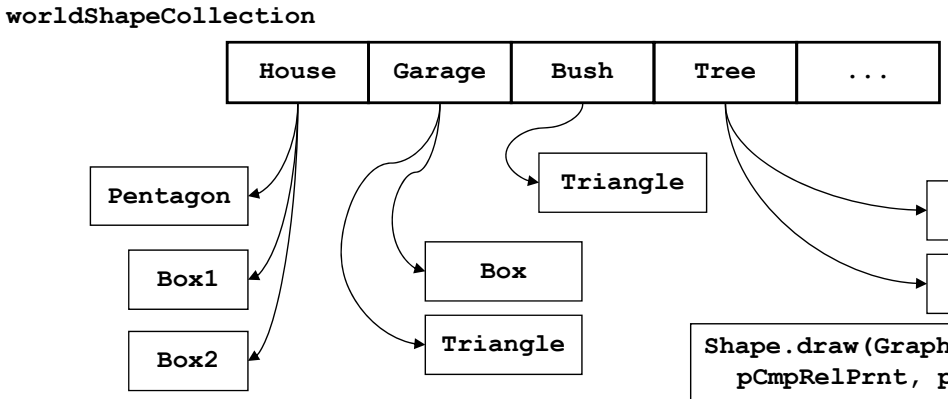
Triangle

Creating A “World”



Tree (Triangle + Box) | Bush (Triangle) | House (Pentagon + Two Boxes) | Garage (Box + Triangle)

The World Object Collection




```
CustomContainer.paint (Graphics g){
  apply "display mapping" and "local
            origin" transforms to g;
  for (Shape s : worldShapeCollection){
    s.draw(g, pCmpRelPrnt, pCmpRelScrn);
  }
  restore xform in g with resetAffine();
}
```

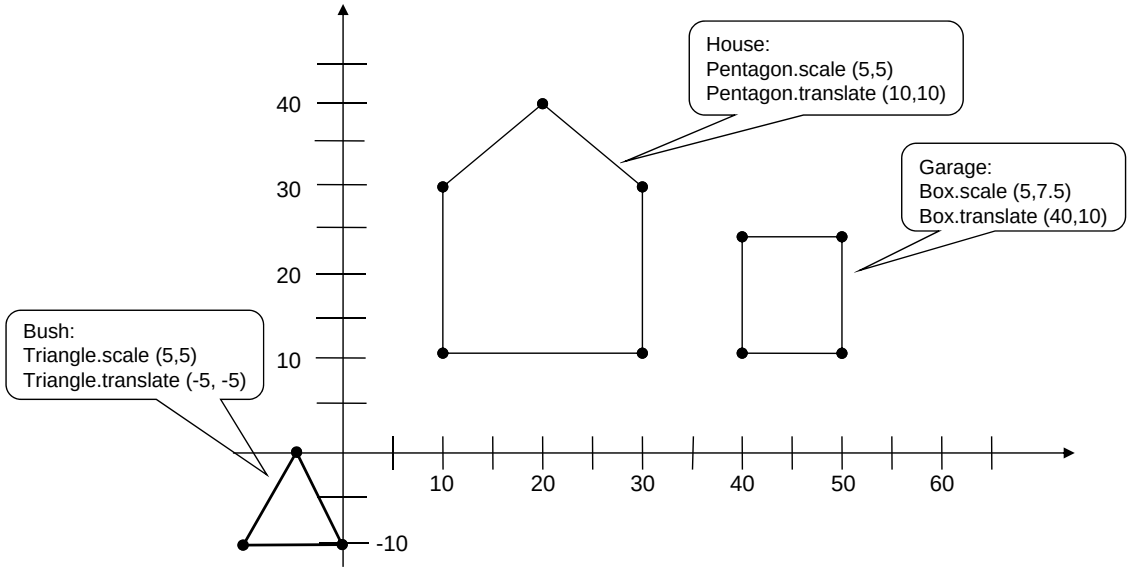
```
Shape.draw(Graphics g, Point
  pCmpRelPrnt, point pCmpRelScrn)
{ save xform in g as gOrigXform;
  apply LTs and "local origin"
                  transforms to g;
  for (each sub shape) {
   sub.draw(g, pCmpRelPrnt,
                  pCmpRelScrn);

  }
 restore xform in g with
      setTranform(gOrigXfrom);
}
```


The World Coordinate System

- "World" ("virtual" or "user") units
  - Independent of display
  - Can represent inches, feet, meters…
- Infinite in all directions
- Object instances are "placed" in the World via *local transformations*

World Coordinate System (cont.)

Example:



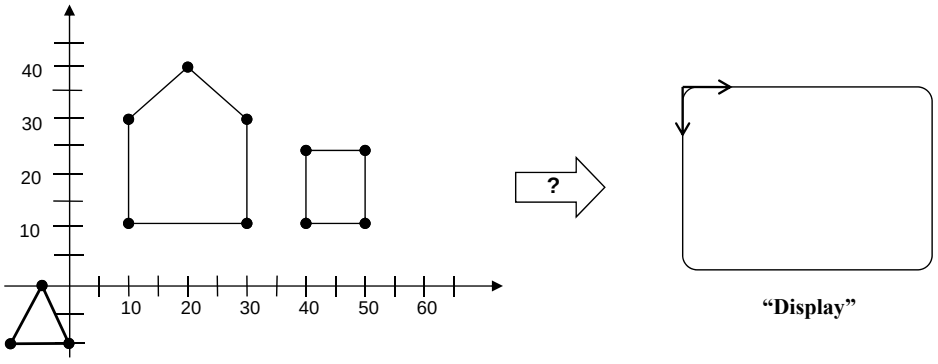
Local Transformations

- With the introduction of world coordinate system, Local Transformations (LTs) no longer place the objects on display, but instead place them in world.
- Hence, in the case of a simple object (e.g., an object which is drawn as a simple triangle) or a top-level object of an hierarchical object (e.g., FireOval object), LTs transform points from local space to world space. Remember that in the previous chapter, LTs were transforming points from local space to display space.
- In case of sub-objects of the hierarchical object (e.g, Flame sub-object of FireOval), just like in the previous chapter, LTs transform points from local space of sub-object to local space of the hierarchical object (apply local scale/rotate/translate to the sub-object to size, orient, and position it relative to the center of the hierarchical object).

Drawing The World On The Display

Needed:

- A way to determine what portion of the (infinite) World gets drawn on the (finite) display
- A "mapping" or *transformation* from *World* to *Display* coordinates



Drawing The World (cont.)

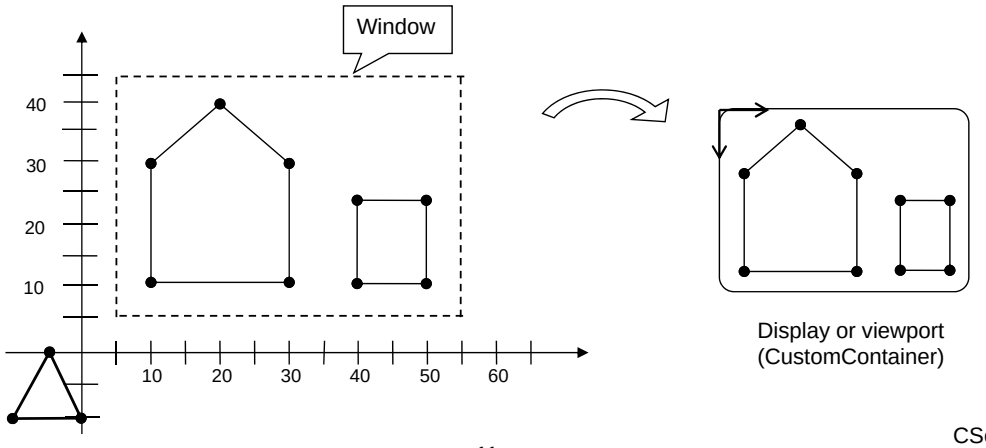
Solution:

- The "Virtual (World) Window"
- A *two-step* mapping through a "*Normalized Device*"

The World "Window"

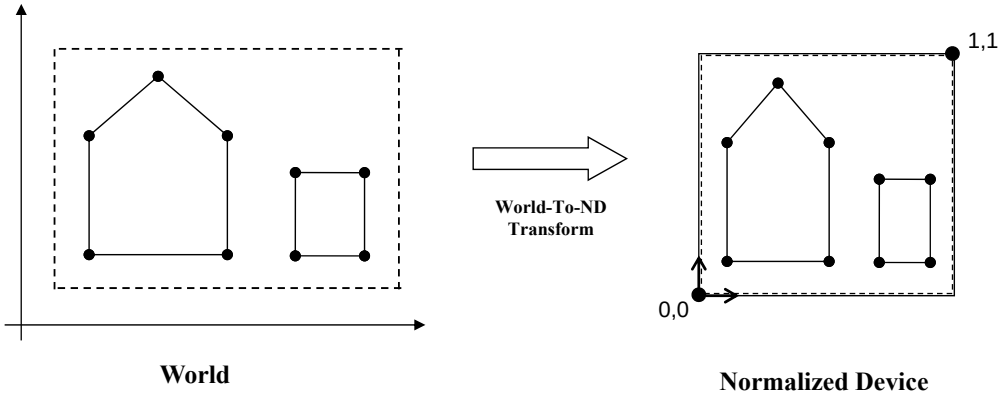
Defines the part of the world that appears on display

- Corners of the window match the corners of the display ("viewport")
- Objects inside window are positioned proportionally in the viewport
- Objects outside window are "clipped" (discarded)



The "Normalized Device"

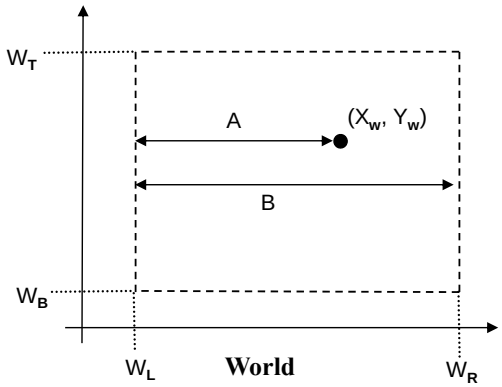
- Properties of the Normalized Device (ND):
  - Square
  - Fractional Coordinates (0.0 .. 1.0)
  - Origin at Lower Left
  - Corners correspond to world window



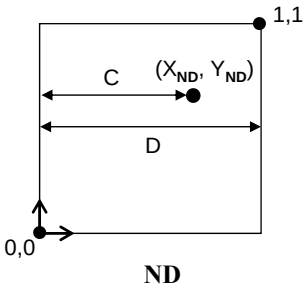
World-To-ND Transform

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} :

$$X_{ND} = \underbrace{(X_w - W_L)}_{\text{A translation}} * \underbrace{\frac{1}{(W_R - W_L)}}_{\text{A scale}}$$

A *translation*

(by -WindowLeft)

A *scale*

(by 1/windowWidth)

Similar rules can be used to derive Y_{ND} :

$$Y_{ND} = \underbrace{(Y_w - W_B)}_{\text{A translation}} * \underbrace{\frac{1}{(W_T - W_B)}}_{\text{A scale}}$$

A *translation*

A *scale*

World-To-ND Transform (cont.)

$X_{ND} = (X_W \bullet \text{Translate}(-W_L)) \bullet \text{Scale}(1 / \text{WindowWidth})$

$Y_{ND} = (Y_W \bullet \text{Translate}(-W_B)) \bullet \text{Scale}(1 / \text{WindowHeight})$

or

$P_{ND} = (P_W \bullet \text{Translate}(-W_L, -W_B)) \bullet \text{Scale}(1/\text{WindowWidth}, 1/\text{WindowHeight})$

- In Matrix Form:

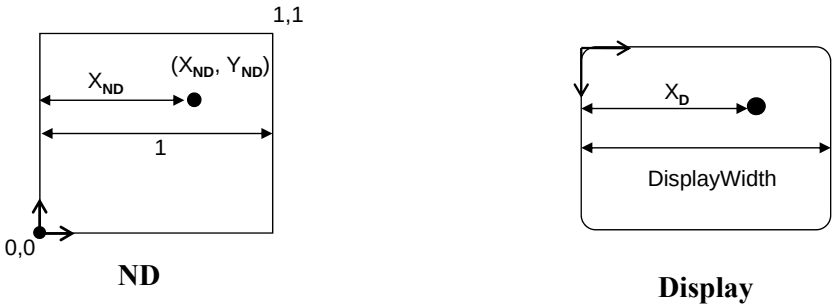
$$\begin{bmatrix} X_{ND} \\ Y_{ND} \\ 1 \end{bmatrix} = \left(\begin{bmatrix} 1/W_w & 0 & 0 \\ 0 & 1/W_h & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -W_L \\ 0 & 1 & -W_B \\ 0 & 0 & 1 \end{bmatrix} \right) \begin{bmatrix} X_W \\ Y_W \\ 1 \end{bmatrix}$$

X

$$\begin{bmatrix} X_{ND} \\ Y_{ND} \\ 1 \end{bmatrix} = \begin{bmatrix} \text{World-to-Normalized-Device (W2ND) Transform} \end{bmatrix} \begin{bmatrix} X_W \\ Y_W \\ 1 \end{bmatrix}$$

ND-To-Display Transform

- A similar approach can be applied

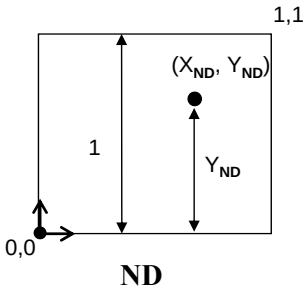


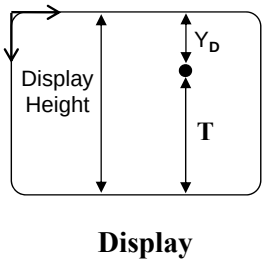
$$\frac{X_{ND}}{1} = \frac{X_D}{DisplayWidth} \; ; \quad \therefore \quad X_D = X_{ND} \times DisplayWidth$$

$$X_D = X_{ND} \bullet Scale(DisplayWidth)$$

ND-To-Display Transform (cont.)

- Similarly for height:





$$\mathbf{T = DisplayHeight - Y_D}$$

$$\frac{Y_{ND}}{1} = \frac{T}{DisplayHeight} = \frac{(DisplayHeight - Y_D)}{DisplayHeight} \;;$$

$$Y_D = (Y_{ND} \times (-DisplayHeight)) + DisplayHeight$$

$$Y_D = (Y_{ND} \bullet Scale(-DisplayHeight)) \bullet Translate(DisplayHeight)$$

ND-To-Display Transform (cont.)

$X_D = (X_{ND} \bullet \text{Scale (DisplayWidth)}) \bullet \text{Translate (0)}$

$Y_D = (Y_{ND} \bullet \text{Scale (-DisplayHeight)}) \bullet \text{Translate (DisplayHeight)}$

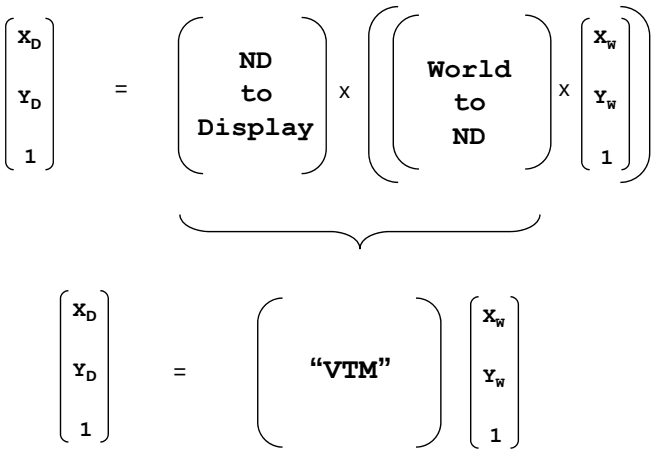
or

$P_D = (P_{ND} \bullet \text{Scale (DisplayWidth, -DisplayHeight)}) \bullet \text{Translate (0, DisplayHeight)}$

- In Matrix Form:

$$\begin{pmatrix} X_D \\ Y_D \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & D_{height} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} D_{width} & 0 & 0 \\ 0 & -D_{height} & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\text{ND-to-Display Transform}} \begin{pmatrix} X_{ND} \\ Y_{ND} \\ 1 \end{pmatrix}$$

Combining Transforms



Using The VTM

- Suppose we have
  - A container with access to a collection of *Shapes*
  - Each shape has a **`draw()`** method which:
    - applies the shape's *local transforms* to gXform
    - calls **`draw()`** on its *sub-shapes*, which applies the sub-shape's local transforms to gXform and draws the sub-shape in "local"coords
- Effect: all draws output *world coordinates*
- We need to *apply the VTM to all output coordinates*

Using The VTM (cont.)

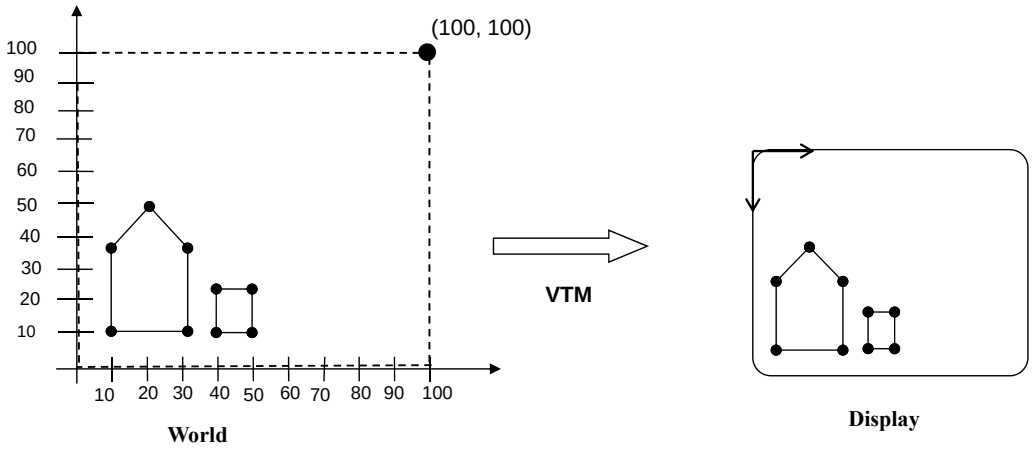
```
public class CustomContainer extends Container {
  Transform worldToND, ndToDisplay, theVTM ;
  private float winLeft, winBottom, winRight, winTop;
  public CustomContainer(){
    //initialize world window
    winLeft = 0;
    winBottom = 0;
    winRight = 931/2; //hardcoded value = this.getWidth()/2 (for the iPad skin)
    winTop = 639/2; //hardcoded value = this.getHeight()/2 (for the iPad skin)
    float winWidth = winRight - winLeft;
    float winHeight = winTop - winBottom;
    //create shapes
    myTriangle = new Triangle((int)(winHeight/5),(int)(winHeight/5));
    myTriangle.translate(winWidth/2, winHeight/2);
    myTriangle.rotate(45);
    myTriangle.scale(1, 2);
    //...[create other simple or hierarchical shapes and add them to collection]
  }
  public void paint (Graphics g) {
    super.paint(g);
    //...[calculate winWidth and winHeight]
    // construct the Viewing Transformation Matrix
    worldToND = buildWorldToNDXform(winWidth, winHeight, winLeft, winBottom);
    ndToDisplay = buildNDToDisplayXform(this.getWidth(), this.getHeight());
    theVTM =  ndToDisplay.copy();
    theVTM.concatenate(worldToND); // worldToND will be applied first to points!
    ... continued ...
```


Using The VTM (cont.)

```
    ... continued ...
    // concatenate the VTM onto the g's current transformation (do not forget to apply "local
    //origin" transformation)
    Transform gXform = Transform.makeIdentity();
    g.getTransform(gXfrom);
    gXform.translate(getAbsoluteX(),getAbsoluteY()); //local origin xform (part 2)
    gXform.concatenate(theVTM); //VTM xform
    gXform.translate(-getAbsoluteX(),-getAbsoluteY()); //local origin xform (part 1)
    g.setTransform(gXform);
    // tell each shape to draw itself using the g (which contains the VTM)
    Point pCmpRelPrnt = new Point(this.getX(), this.getY());
    Point pCmpRelScrn = new Point(getAbsoluteX(),getAbsoluteY());
    for (Shape s : shapeCollection)
      s.draw(g, pCmpRelPrnt, pCmpRelScrn);
    g.resetAffine() ;
  }
  private Transform buildWorldToNDXform(float winWidth, float winHeight, float
  winLeft, float winBottom){
    Transform  tmpXfrom = Transform.makeIdentity();
    tmpXfrom.scale( (1/winWidth) , (1/winHeight) );
    tmpXfrom.translate(-winLeft,-winBottom);
    return tmpXfrom;
  }
  private Transform buildNDToDisplayXform (float displayWidth, float displayHeight){
    Transform tmpXfrom = Transform.makeIdentity();
    tmpXfrom.translate(0, displayHeight);
    tmpXfrom.scale(displayWidth, -displayHeight);
    return tmpXfrom;
  }
//...[other methods of CustomContainer]
}//end of CustomContainer
```

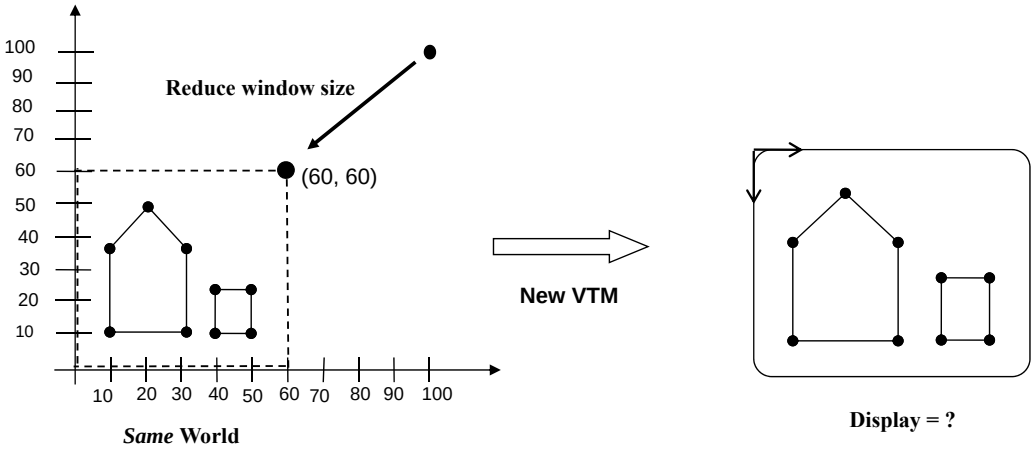
Changing the Window Size

Suppose we start with this:



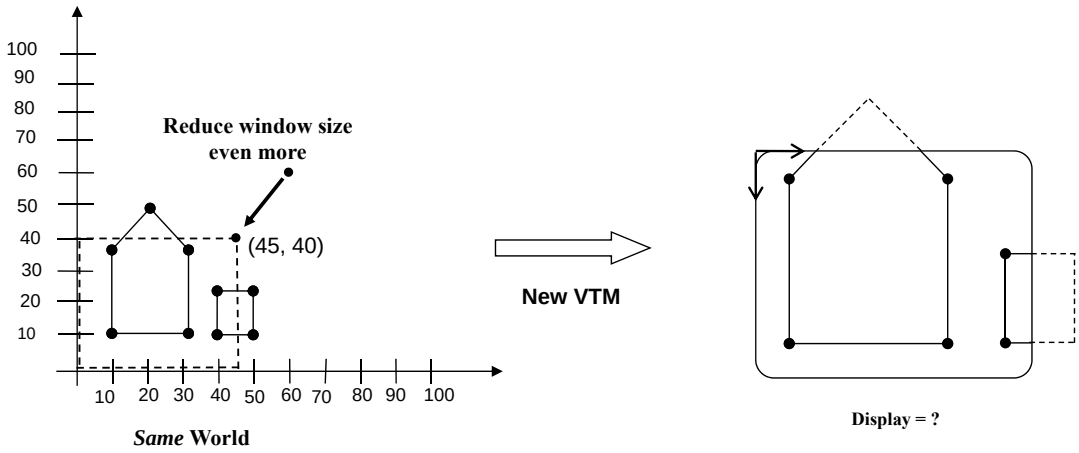
Changing the Window Size (cont.)

Now we change window size,
recompute the VTM, and repaint



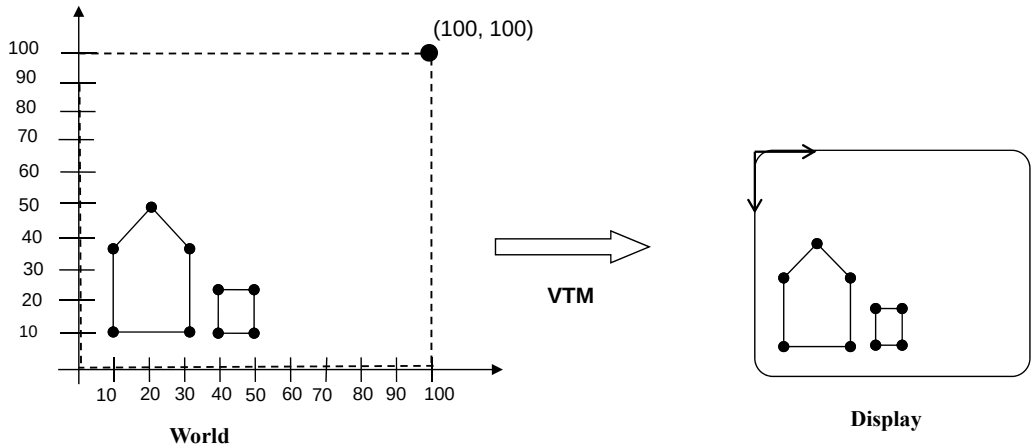
Changing the Window Size (cont.)

- Now we change window size *more*, recompute the VTM, and repaint again



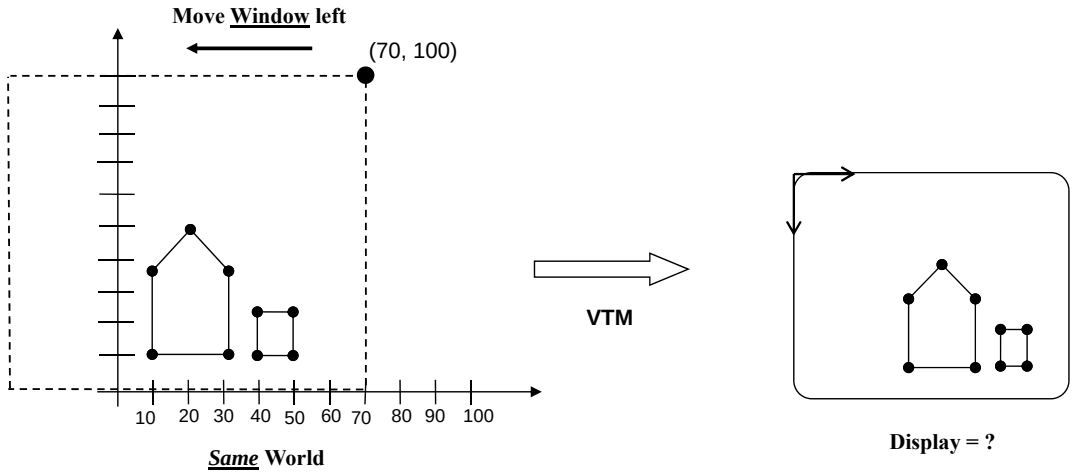
Changing Window *Location*

Suppose we start with this:



Changing the Window Location (cont.)

Now we change window *location*,
recompute the VTM, and repaint



Adding Zoom and Pan Functionality

```
/* Following methods should be added to CustomContainer to allow zooming and panning */
public void zoom(float factor) {
    //positive factor would zoom in (make the worldWin smaller), suggested value is 0.05f
    //negative factor would zoom out (make the worldWin larger), suggested value is -0.05f
    //...[calculate winWidth and winHeight]
    float newWinLeft = winLeft + winWidth*factor;
    float newWinRight = winRight - winWidth*factor;
    float newWinTop = winTop - winHeight*factor;
    float newWinBottom = winBottom + winHeight*factor;
    float newWinHeight = newWinTop - newWinBottom;
    float newWinWidth = newWinRight - newWinLeft;
    //in CN1 do not use world window dimensions greater than 1000!!!
    if (newWinWidth <= 1000 && newWinHeight <= 1000 && newWinWidth > 0 && newWinHeight > 0 ){
        winLeft = newWinLeft;
        winRight = newWinRight;
        winTop = newWinTop;
        winBottom = newWinBottom;
    }
    else
        System.out.println("Cannot zoom further!");
    this.repaint();
}
public void panHorizontal(double delta) {
    //positive delta would pan right (image would shift left), suggested value is 5
    //negative delta would pan left (image would shift right), suggested value is -5
    winLeft += delta;
    winRight += delta;
    this.repaint();
}
public void panVertical(double delta) {
    //positive delta would pan up (image would shift down), suggested value is 5
    //negative delta would pan down (image would shift up), suggested value is -5
    winBottom += delta;
    winTop += delta;
    this.repaint();
}
```

Zoom with Pinching in CN1

`Component` build-in class has **`pinch()`** method. You can override it to call the zoom() method in the previous slide to zoom whenever the user pinches the display (the user's two fingers come "closer" together or go "away" from each other on display).

The simulator assumes one finger is always at the screen origin (the top left corner of the screen), hence:

"closer" pinching (zooming out) is simulated by simultaneous right mouse click and mouse movement towards the screen origin.

"away" pinching (zooming in) is simulated by simultaneous right mouse click and mouse movement going away from the screen origin.

Zoom with Pinching in CN1 (cont.)

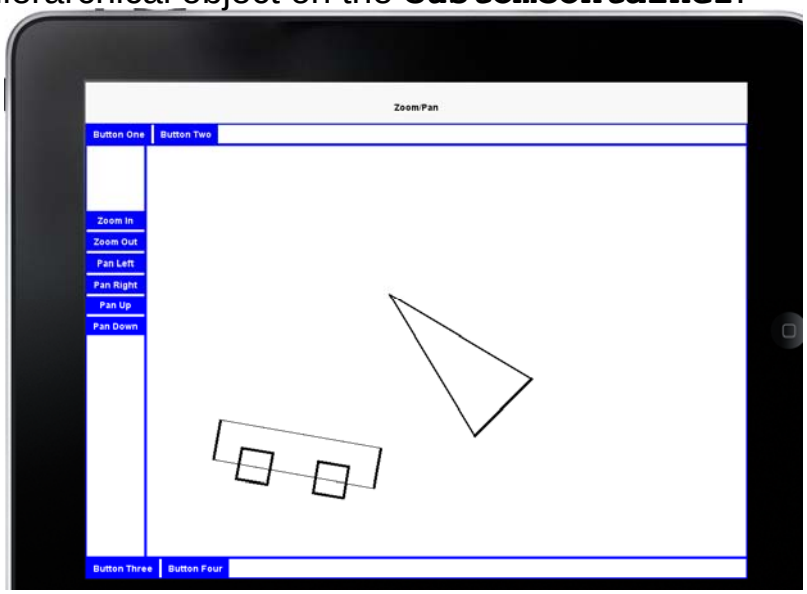
```
/* Override pinch() in CustomContainer to allow zooming with pinching*/
@Override
public boolean pinch(float scale){
    if(scale < 1.0){
        //Zooming Out: two fingers come closer together (on actual device), right mouse
        //click + drag towards the top left corner of screen (on simulator)
        zoom(-0.05f);
    }else if(scale>1.0){
        //Zooming In: two fingers go away from each other (on actual device), right mouse
        //click +  drag away from the top left corner of screen (on simulator)
        zoom(0.05f);
    }
    return true;
}
```

Pan with Pointer Dragging in CN1

```
    /* Override pointerDrag() in CustomContainer to allow panning with a pointer drag which is
simulated with a mouse drag (i.e., simultaneous mouse left click and mouse movement). Below
code moves the world window in the direction of dragging (e.g., dragging the pointer towards
left and top corner of the display would move the object towards the right and top corner of
the display) */
    private Point pPrevDragLoc = new Point(-1, -1);
    @Override
    public void pointerDragged(int x, int y)
    {
    if (pPrevDragLoc.getX() != -1)
    {
        if (pPrevDragLoc.getX() < x)
            panHorizontal(5);
        else if (pPrevDragLoc.getX() > x)
            panHorizontal(-5);
        if (pPrevDragLoc.getY() < y)
            panVertical(-5);
        else if (pPrevDragLoc.getY() > y)
            panVertical(5);
    }

    pPrevDragLoc.setX(x);
    pPrevDragLoc.setY(y);
    }
```

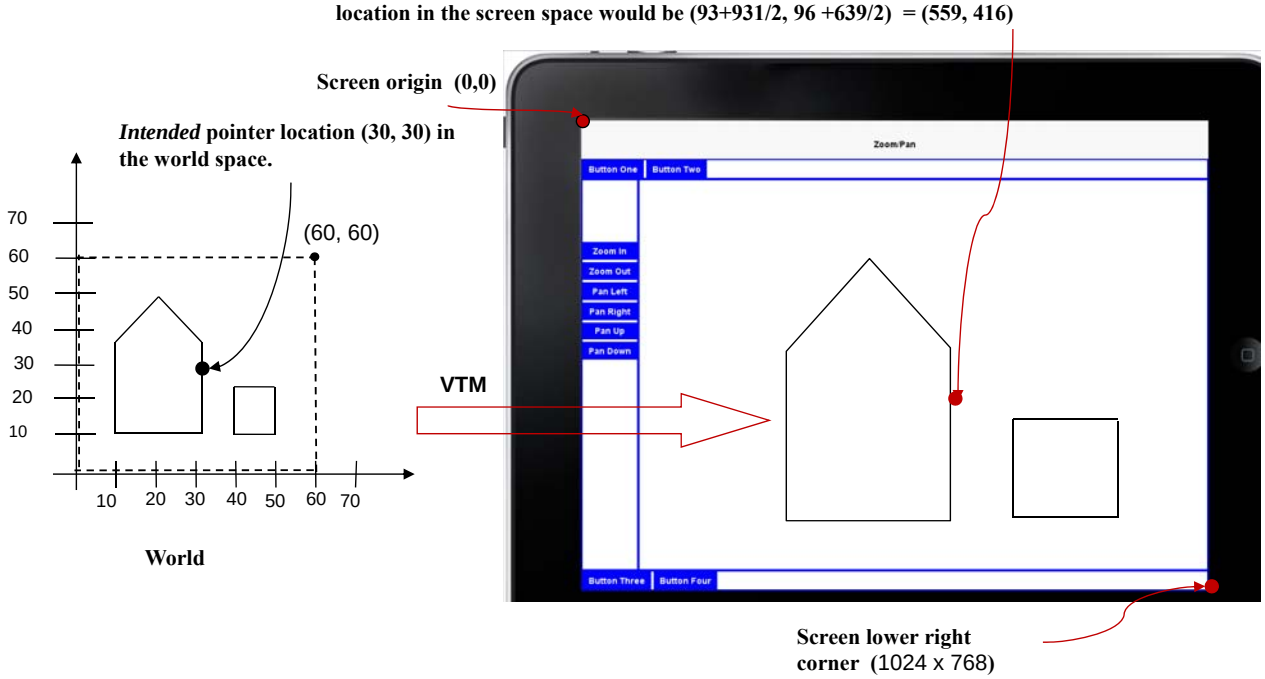

Zoom/Pan App ScreenShot

Create a form with a border layout and put the **`CustomContainer`** object to the center. Call zoom and pan methods of **`CustomContainer`** (with proper parameter values) when the buttons on the west container are clicked and when pinching and pointer dragging happen. In addition to a triangle, draw a hierarchical object on the **`CustomContainer`**.



Mapping Pointer To World Coords

Pointer click location: assume it is at exactly the middle of the display (CustomContainer) Since myCustomContainer.getAbsoluteX() = 93, myCustomContainer.getAbsoluteY() = 96 myCustomContainer.getWidth() = 931, myCustomContainer.getHeight() =639 then the pointer location in the screen space would be (93+931/2, 96 +639/2) = (559, 416)



Mapping Pointer To World Coords

When ***drawing*** (outputting) points, we apply the following transform:

$$\begin{bmatrix} X_D \\ Y_D \\ 1 \end{bmatrix} = \begin{bmatrix} & VTM & \end{bmatrix} \times \begin{bmatrix} X_w \\ Y_w \\ 1 \end{bmatrix}$$

Display Point **World Point**

(Note that, in CN1, as usual, we also apply "local origin" transformations before and after applying VTM)

Mapping Pointer To World Coords (cont.)

- In CN1, from **`pointerPressed()`**, we get the pointer location relative to the *Screen* orgin. We make this point relative to the *Display* origin by deducting **`getAbsoluteX()/Y()`** value of the display.
- Then we can calculate the corresponding point in the world space. We need to go “backwards”:

$$\begin{bmatrix} X_w \\ Y_w \\ 1 \end{bmatrix} = \begin{bmatrix} ?? \end{bmatrix} \times \begin{bmatrix} X_D \\ Y_D \\ 1 \end{bmatrix}$$

World Point **Display Point (given)**

Mapping Pointer To World Coords (cont.)

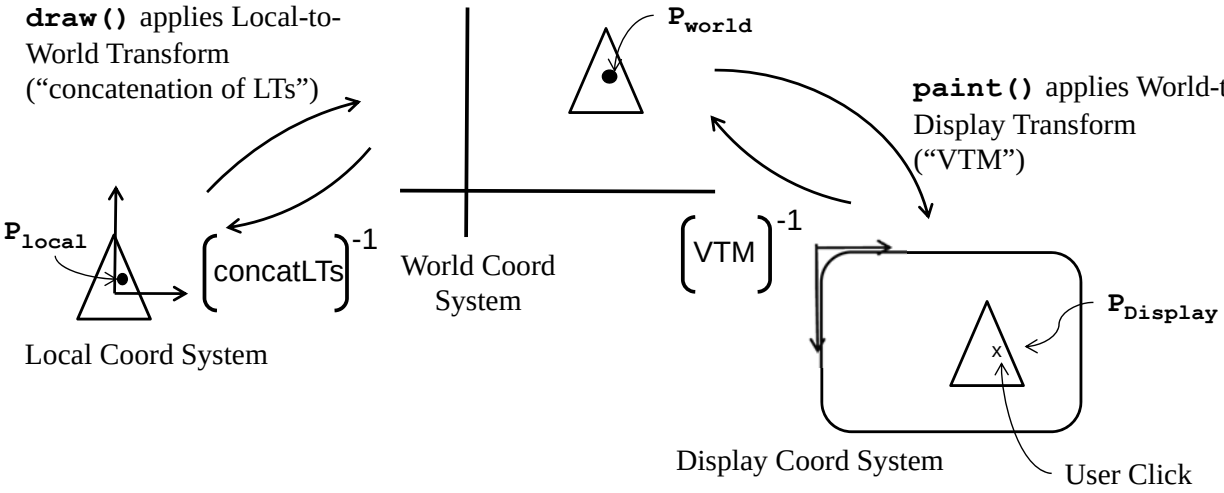
- **We need the *inverse* of the VTM**

$$\begin{bmatrix} X_w \\ Y_w \\ 1 \end{bmatrix} = \begin{bmatrix} VTM \end{bmatrix}^{-1} \times \begin{bmatrix} X_D \\ Y_D \\ 1 \end{bmatrix}$$

World Point **Display Point**

```
Transform theVTM, inverseVTM ;
// ...code here to define the contents of theVTM.
inverseVTM = Transform.makeIdentity();
try {
    theVTM.getInverse(inverseVTM);
} catch (NotInvertibleException e) {
    System.out.println("Non invertible xform!");}
inverseVTM.transformPoint(fPtr, fPtr); //fPtr is a float array that first holds the
    //point in the display space, then it is transformed into the world space.
```

Selection / Containment



To generate the concatLTs, same order of transformations used in draw() must be used.

If the object is a hierarchical object the sub-shapes should also apply inverse of their concatLTs to see if they contain the point.

Selection of Hierarchical Objects

```
/* Create a hierarchical object in the constructor of CustomContainer */
public class CustomContainer extends Container {
 myHierObj = new HierObj((int)(winHeight/5));
 myHierObj.translate(winWidth/4, winHeight/4);
 myHierObj.rotate(-10);
 myHierObj.scale(2, 1);
 //...[rest of the constructor code]
}
/* Also, override the pointerPressed() in CustomContainer to get the pointer location in screen
space */
@Override
public void pointerPressed(int x, int y){
 //(x, y) is the pointer location relative to screen origin
 //make it relative to display origin
 float [] fPtr = new float [] {x - getAbsoluteX(), y - getAbsoluteY()};
 Transform inverseVTM = Transform.makeIdentity();
 try {
  theVTM.getInverse(inverseVTM);
 } catch (NotInvertibleException e) {
  System.out.println("Non invertible xform!");}
 //calculate the location of fPtr the in world space
 inverseVTM.transformPoint(fPtr, fPtr);
 if (myHierObj.contains(fPtr))
  myHierObj.setSelected(true);
 else
  myHierObj.setSelected(false);
 repaint();
}
```

Selection of Hierarchical Objects (cont.)

```
/* The constructor of the hierarchical object build the object from three sub-objects (all of which are
based on a "square" primitive). */
public HierObj(int size) {
 // create an array to hold the sub-objects
 sobjs = new Square[3];
 Square body = new Square(size); body.scale(1.0f, 0.5f); sobjs[0] = body;
 Square leg0 = new Square(size/5); leg0.scale(1.0f, 2f);
 leg0.translate(-size/4, -size/4); sobjs[1] = leg0;
 Square leg1 = new Square(size/5); leg1.scale(1.0f, 2f);
 leg1.translate(size/4, -size/4); sobjs[2] = leg1;
 //...[rest of the constructor code]}
/* contains() of HierObj apply inverse concatLTs to fPtr and call sub-objects's contains() methods*/
public boolean contains(float[] fPtr) {//signature of contains() in ISelectable has to be updated
 //concatenate all LTs (make sure to follow the same transformation order used in draw())
 Transform concatLTs = Transform.makeIdentity();
 concatLTs.translate(myTranslation.getTranslateX(), myTranslation.getTranslateY());
 concatLTs.concatenate(myRotation);
 concatLTs.scale(myScale.getScaleX(), myScale.getScaleY());
 //calculate inverse of concatLTs
 Transform inverseConcatLTs = Transform.makeIdentity();
 try {
  concatLTs.getInverse(inverseConcatLTs);
 } catch (NotInvertibleException e) {
  System.out.println("Non invertible xform!");}
 //fPtr is in the world space, calculate the corresponding point in the local space of HierObj
 inverseConcatLTs.transformPoint(fPtr, fPtr);
 for (Square sobj : sobjs) {
  //make sure that the point is not already transformed with previous sub-object's inverse concatLTs
  float[] fPtrCopy =  new float[] {fPtr[0], fPtr[1]};
  if ( sobj.contains(fPtrCopy))
    return true;
  }
 return false;}
```


Selection of Hierarchical Objects (cont.)

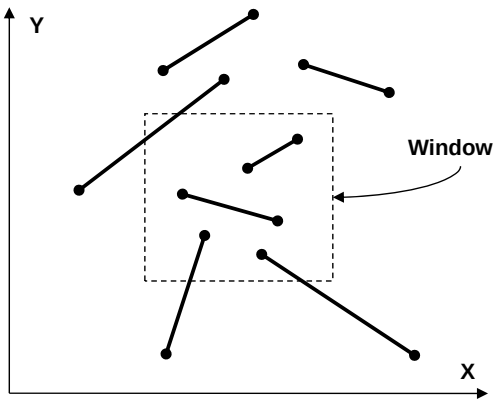
```
/* setSelected() of HierObj call sub-object's setSelected() methods, so that if a sub-object of HierObj
gets selected/unselected, all sub-objects would get selected/unselected */
public void setSelected(boolean b) {
 for (Square sobj : sobjs)
  sobj.setSelected(b);
}
/* The constructor of the square define the object points in local space */
public Square(int givenSize) {
 size = givenSize;
 lowerLeftInLocalSpace = new Point(-size/2, -size/2); //corresponds to upper left corner on screen
 //...[rest of the constructor code]
}
/* contains() of Square apply inverse concatLTs to fPtr and checks if the point is inside*/
public boolean contains(float[] fPtr) {
 //concatenate all LTs (make sure to follow the same order used in draw())
 Transform concatLTs = Transform.makeIdentity();
 concatLTs.translate(myTranslation.getTranslateX(), myTranslation.getTranslateY());
 concatLTs.concatenate(myRotation);
 concatLTs.scale(myScale.getScaleX(), myScale.getScaleY());
 Transform inverseConcatLTs = Transform.makeIdentity();
 try {  concatLTs.getInverse(inverseConcatLTs);
 } catch (NotInvertibleException e) {
  System.out.println("Non invertible xform!");}
 //fPtr is in the local space of HierObj, calculate the corresponding point in the local space of Square
 inverseConcatLTs.transformPoint(fPtr, fPtr);
 int px = (int)fPtr[0]; //pointer location relative to
 int py = (int)fPtr[1]; //local origin
 int xLoc = lowerLeftInLocalSpace.getX(); //square lower left corner
 int yLoc = lowerLeftInLocalSpace.getY(); //location relative to local origin
 if ( (px >= xLoc) && (px <= xLoc+size)  && (py >= yLoc) && (py <= yLoc+size) )
  return true;
 else
  return false;}
```

Selection of Hierarchical Objects (cont.)

```
/* setSelected() of Square would set the bSelected class field as before...*/
public void setSelected(boolean b) {
 bSelected = b;
}
/* also, draw() of Square would draw the shape by checking bSelected as before...*/
public void draw(Graphics g,Point pCmpRelPrnt, Point pCmpRelScrn) {
 g.setColor(ColorUtil.BLACK);
 Transform gXform = Transform.makeIdenitity();
 g.getTransform(gXform);
 Transform gOrigXform = gXform.copy();
 //apply LTs (also "local origin" transformations)
 gXform.translate(pCmpRelScrn.getX(),pCmpRelScrn.getY());
 gXform.translate(myTranslation.getTranslateX(), myTranslation.getTranslateY());
 gXform.concatenate(myRotation);
 gXform.scale(myScale.getScaleX(), myScale.getScaleY());
 gXform.translate(-pCmpRelScrn.getX(),-pCmpRelScrn.getY());
 g.setTransform(gXform);
 //draw the shape
 if (bSelected)
  g.fillRect(pCmpRelPrnt.getX()+lowerLeftInLocalSpace.getX(),
             pCmpRelPrnt.getY()+lowerLeftInLocalSpace.getY(), size, size);
 else
  g.drawRect(pCmpRelPrnt.getX()+lowerLeftInLocalSpace.getX(),
             pCmpRelPrnt.getY()+lowerLeftInLocalSpace.getY(), size, size);
 g.setTransform(gOrigXform); //restore the original xform
}
```


Clipping

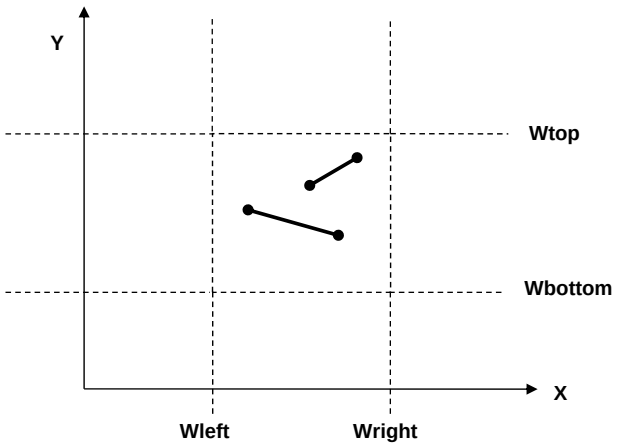
- **Need to suppress output that lies outside the window**
- **For lines, various possibilities:**
  - Both endpoints inside (totally visible)
  - One point inside, the other outside (partially visible)
  - Both endpoints outside (totally invisible ?)



Visibility Tests

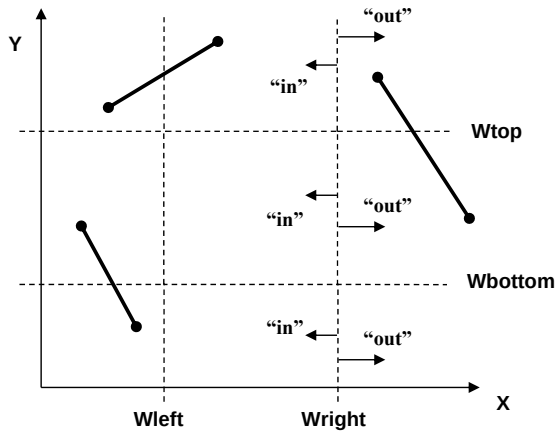
- **"Trivial Acceptance"**
  - Line is completely visible if both endpoints are:

```
Below Wtop && Above Wbottom && rightOf Wleft && leftOf Wright
```



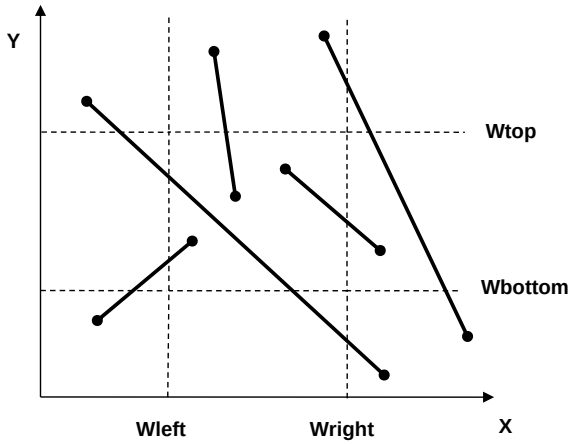
Visibility Tests (cont.)

- **"Trivial Rejection"**
  - Line is completely invisible if both endpoints are on the "out" side of any window boundary



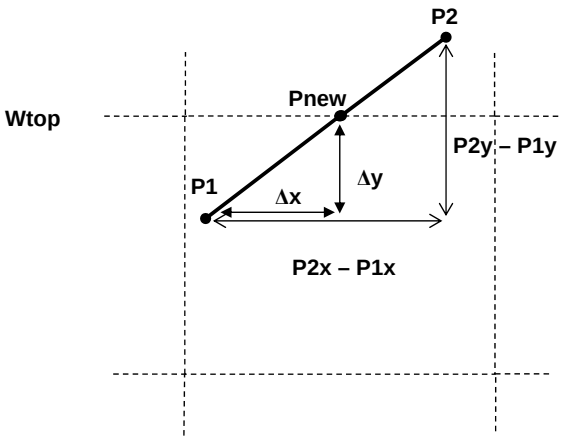
Visibility Tests (cont.)

- Some cases cannot be trivially accepted or rejected :



Clipping Non-Trivial Lines

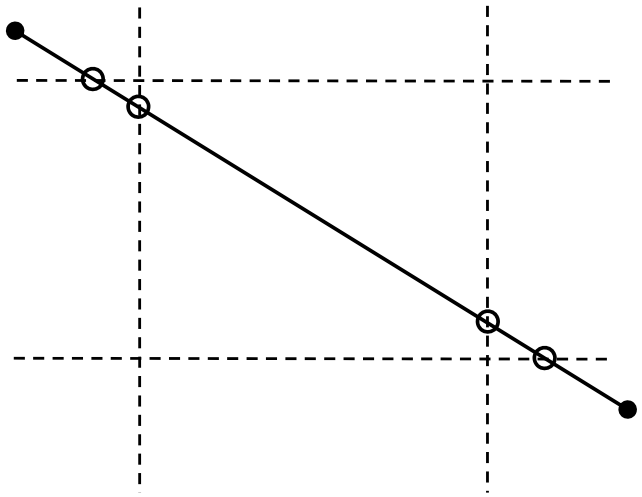
- At least ONE endpoint will be OUTSIDE
  - Compute intersection with (some) boundary
  - Replace “outside” point with Intersection point
  - Repeat as necessary (i.e. until acceptance or empty)




```
Slope = (P2y-P1y) / (P2x-P1x)

PnewY = Wtop

Δy / Δx = Slope

Δy = PnewY – P1y

Δx = Δy / Slope

PnewX = P1x + Δx
```

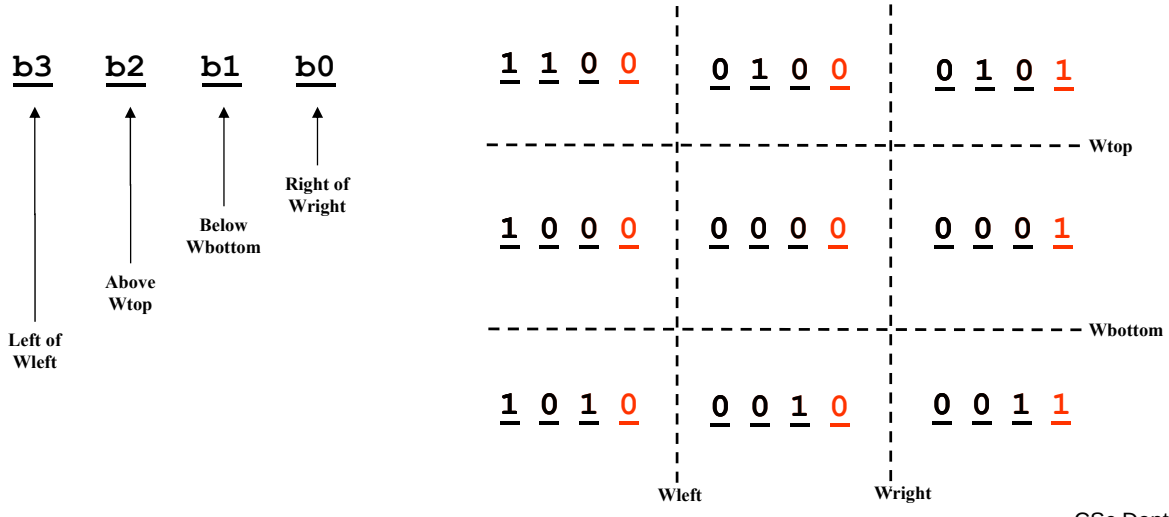
Clipping Non-Trivial Lines (cont.)

- 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



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 = windowTop ;
  } 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 = windowBottom ;
  } else if (p1 is left of the window) {
     // find intersect of p1-p2 with window LEFT side
     x1 = windowLeft  ;
     y1 = intersectWithLeftside (p1,p2)      // get the y-intersect
  } else if (p1 is right of the window) {
     // find intersection with RIGHT side
     x1 = windowRight ;
     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) );
}
```

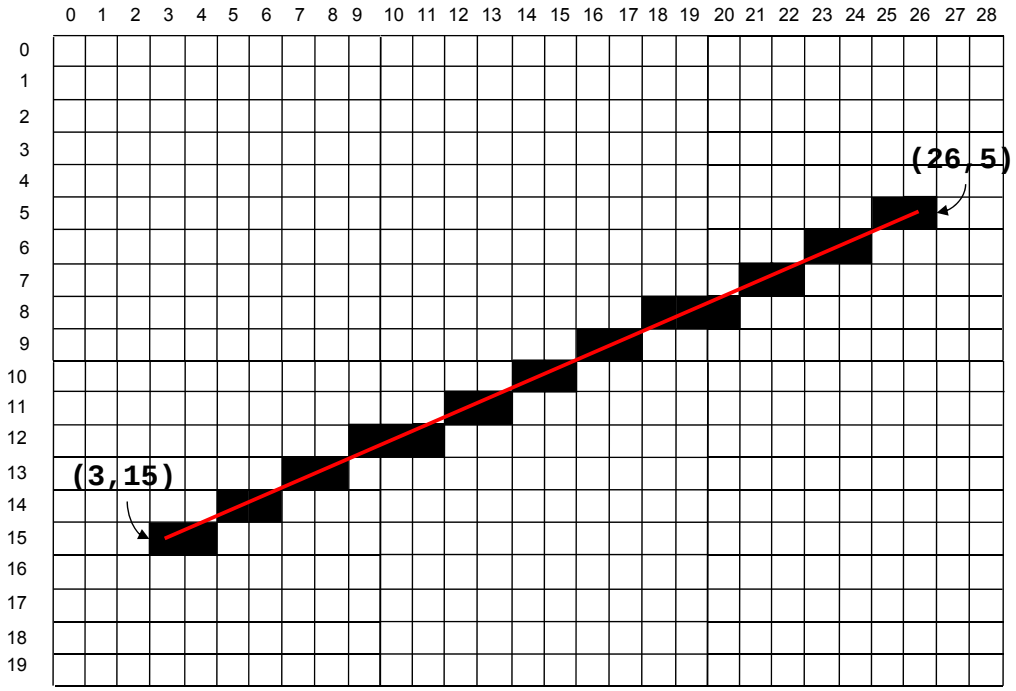
16 - Lines and Curves

Computer Science Department
California State University, Sacramento

Overview

- Rasterization
- Line-based Graphical Primitives
- Parametric Line Representation
- Quadratic & Cubic Bezier Curves
  - Geometric and analytical definitions
- Rendering Via Recursive Subdivision
- Applications of Curves

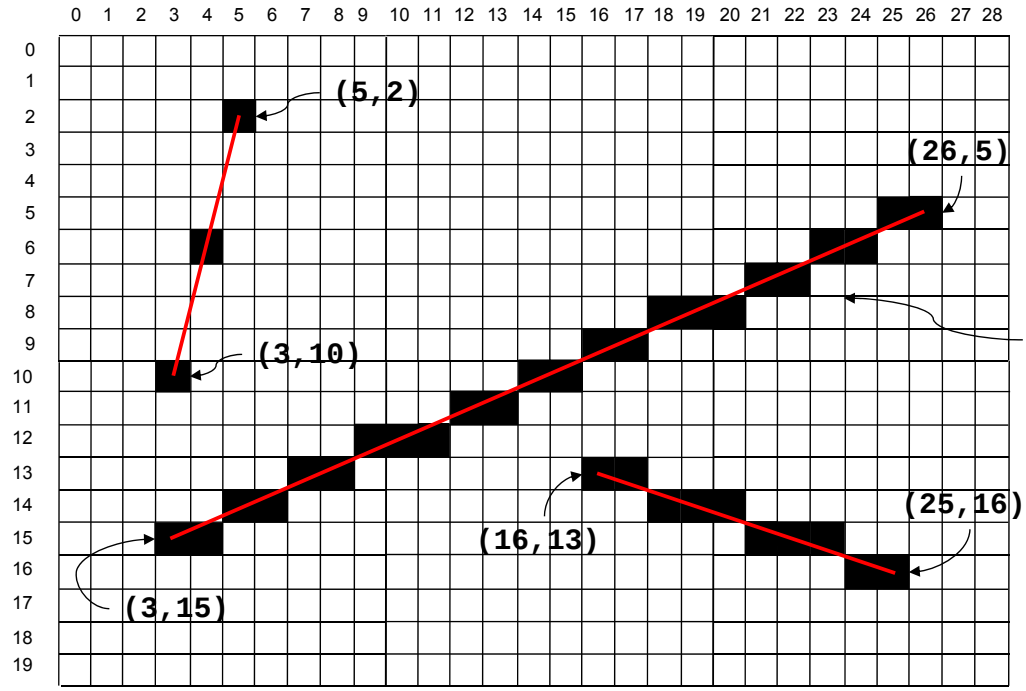
Rasterization



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 absolute value of the
 *  slope of the line is < 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



slope =
(5-15)/(26-3) =
-10/23 =
-0.43

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 factor ;            // denominator used in xIncr and yIncr formulas
  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(dy/dx) < 1 then
    factor = abs (dx)     // if abs(slope) < 1, to take unit steps in X, factor = abs(dx)= dx
  else
    factor = abs (dy) ;   // if abs(slope) >= 1, to take unit steps in Y, factor = abs(dy)
  xIncr = dx / factor ;   // increase in X per step. If abs(slope)<1, xIncr = 1. If
                          // abs(slope)>=1, xIncr = 1/abs(slope)= abs(dx)/abs(dy) = dx/abs(dy)
  yIncr = dy / factor ;   // increase in Y per step. If abs(slope)>=1, yIncr = 1 (if slope is
                          // positive) OR yIncr = -1 (if slope is negative). If abs(slope)<1,
                          // yIncr = slope = dy/dx = dy/abs(dx)
  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) ;
  }
}
```

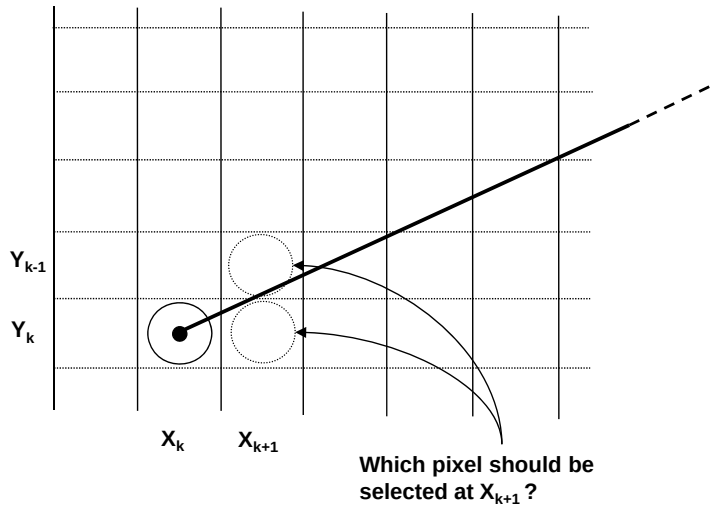
Problem with DDA Algorithm

- In the for-loop located at the end of algorithm it does a floating point arithmetic:
  - It is expensive when repeated many times.
  - It can cause a floating point error.

- These problems can result is highly inaccurate rasterization results.

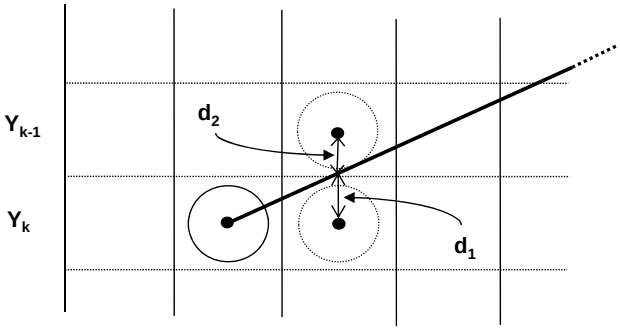
The "Pixel Selection" Decision

- Basic question: which is the best "next pixel"?



The "Pixel Decision" Parameter

- Choose the pixel *closest to the true line*



Same as "sign(d1-d2) is +"

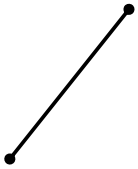
```
if ((d1-d2) > 0)
   choose pixel Y_{k-1}
else
   choose pixel Y_k
```


Bresenham's Algorithm

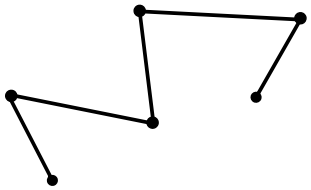
- Bresenham [IBM, 1962] figured out how to make the "sign(d1-d2) is positive" test using only integer arithmetic.
- No floating point involved!
- This results in rasterization that is at the same time faster and also more accurate (because it always chooses the "best next pixel").

Graphical Primitives

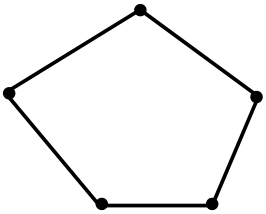
- Point- and Line-based



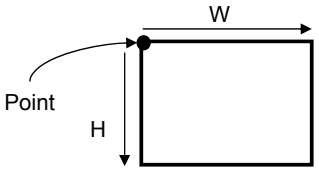
Line



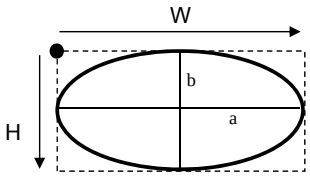
"Polyline"



"Polygon"



Rectangle

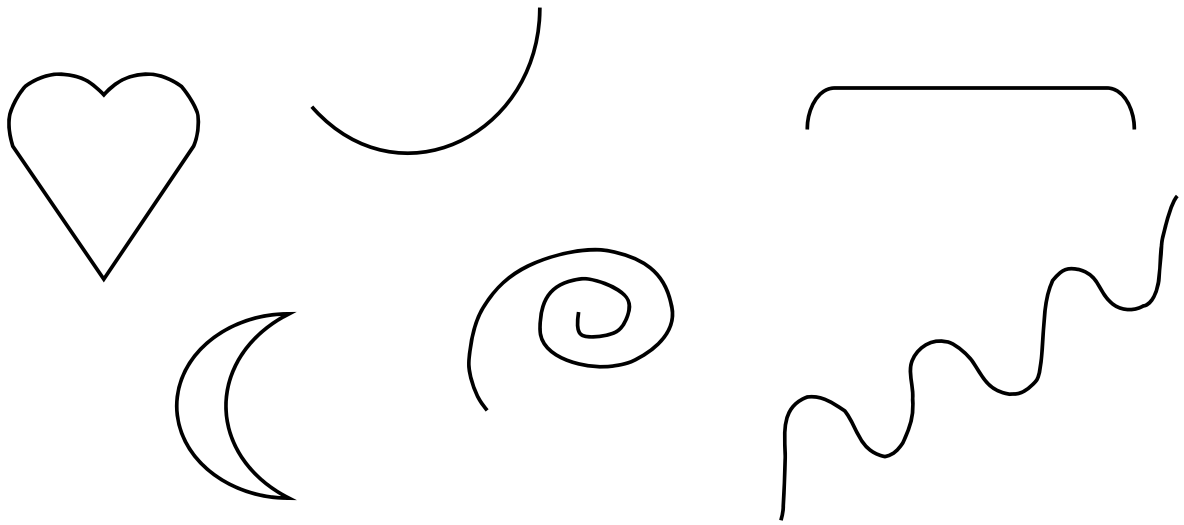


Oval

$$\frac{(x - xCenter)^2}{a^2} + \frac{(y - yCenter)^2}{b^2} = 1$$

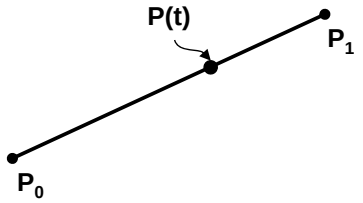
Curves Of Higher Complexity

- What if we want to draw shapes like these?



Parametric Line Representation

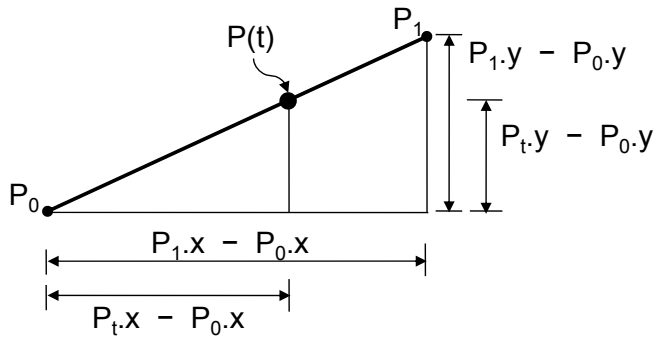
- Lines can be represented in terms of known quantities in several ways :
  - **`Y = mX + b`**// line with slope "m" and Y-intercept "b"
  - **`(P0, P1)`** // line containing $\mathbf{P_0}$ and $\mathbf{P_1}$
- Any point on **(P_0, P_1)** can be represented with a single *parameter value* '***t***'



- **'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:



$$t = \frac{P_t - P_0}{P_1 - P_0}$$

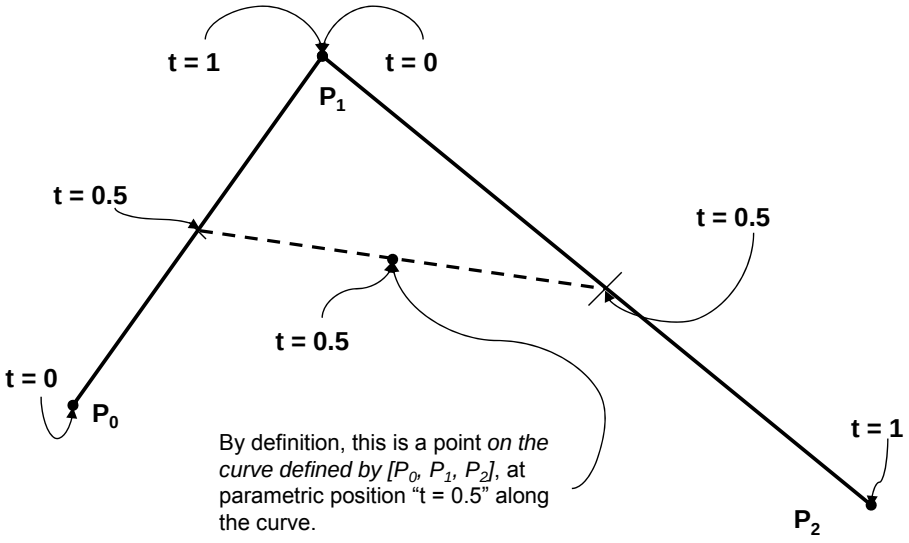
$$t\left(P_1 - P_0\right) = P_t - P_0$$

$$P_t = P_0 + t\left(P_1 - P_0\right)$$

$$P_t = \left(1 - t\right) P_0 + t\, P_1$$

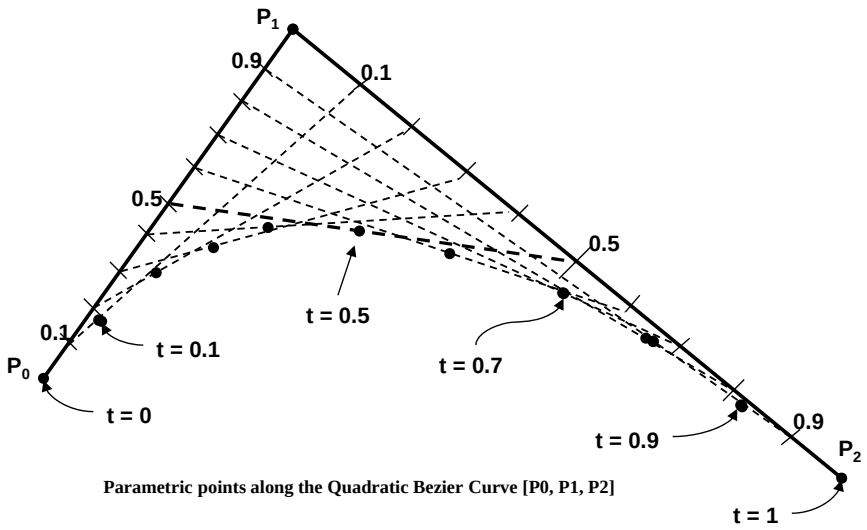
Quadratic Bezier Curves

- Geometric description



Quadratic Bezier Curves (cont.)

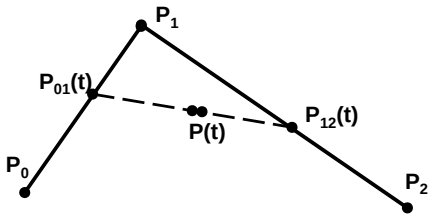
- Connecting points of equal parametric value generates a curve:



Parametric points along the Quadratic Bezier Curve [P0, P1, P2]

Quadratic Bezier Curves (cont.)

- Analytic definition



$$P_{01}(t) = t \cdot P_1 + (1-t) \cdot P_0 \quad [1]$$

and

$$P_{12}(t) = t \cdot P_2 + (1-t) \cdot P_1 \quad [2]$$

and a point on the curve $[P_0 \quad P_1 \quad P_2]$ is defined as

$$P(t) = t \cdot (P_{12}(t)) + (1-t) \cdot (P_{01}(t)) \quad [3]$$

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

$$P(t) = (1-t)^2 \cdot P_0 + (-2t^2 + 2t) \cdot P_1 + (t^2) \cdot P_2$$

Curves as Weighted Sums

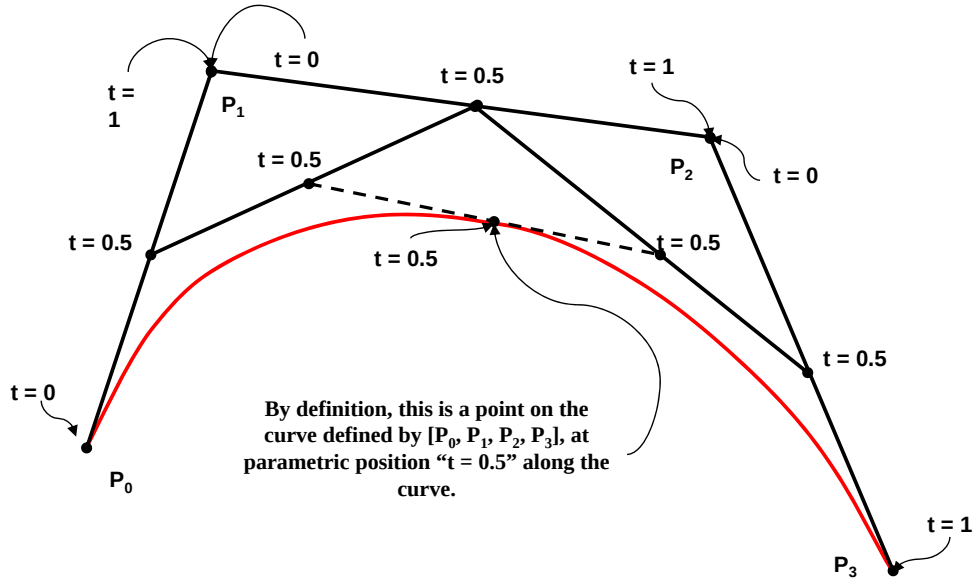
$$P(t) = (1-t)^2 \cdot P_0 + (-2t^2 + 2t) \cdot P_1 + (t^2) \cdot P_2$$

$$P(t) = \sum_{i=0}^{2} P_i \cdot B_i(t), \text{where} \begin{cases} B_0(t) = (1-t)^2 \\ B_1(t) = (-2t^2 + 2t) \\ B_2(t) = t^2 \end{cases}$$

- *A point on the curve is a weighted sum of the three "control points"*
  - The "weightings" are the quadratic polynomials, evaluated at "*t* "

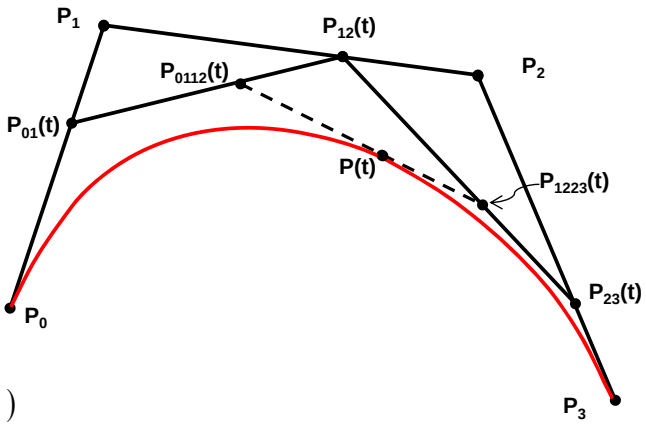
Cubic Bezier Curves

- Geometric description



Cubic Bezier Curves (cont.)

- Analytic definition



$$P_{01}(t) = t \cdot P_1 + (1-t) \cdot P_0$$

$$P_{12}(t) = t \cdot P_2 + (1-t) \cdot P_1$$

$$P_{23}(t) = t \cdot P_3 + (1-t) \cdot P_2$$

$$P_{0112}(t) = t \cdot P_{12}(t) + (1-t) \cdot P_{01}(t)$$

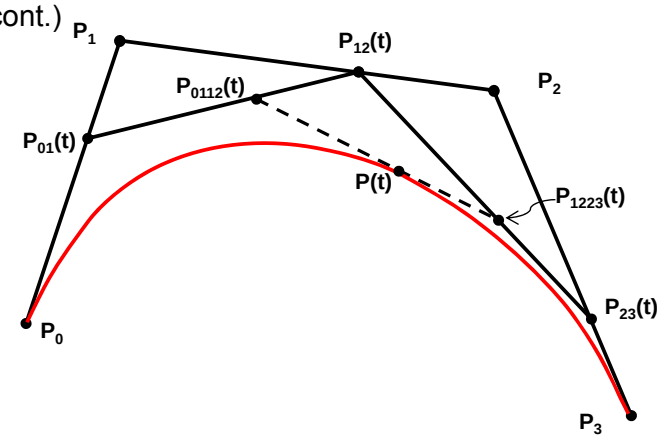
$$P_{1223}(t) = t \cdot P_{23}(t) + (1-t) \cdot P_{12}(t)$$

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.)

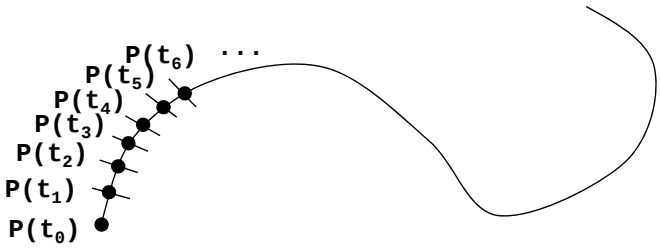


$$P(t) = t\cdot(P_{1223}(t)) + (1-t)\cdot(P_{0112}(t))$$
$$= \underline{(1-t)^3}\cdot P_0 + \underline{(3t^3-6t^2+3t)}\cdot P_1 + \underline{(-3t^3+3t^2)}\cdot P_2 + \underline{(t^3)}\cdot P_3$$
$$= \sum_{i=0}^{3} P_i \cdot B_{i,3}(t)$$

Drawing Bezier Curves

- Iterative approach

```
moveTo (P(t0));
drawTo (P(t1));
drawTo (P(t2));
drawTo (P(t3));
...
```



Drawing Bezier Curves (cont.)

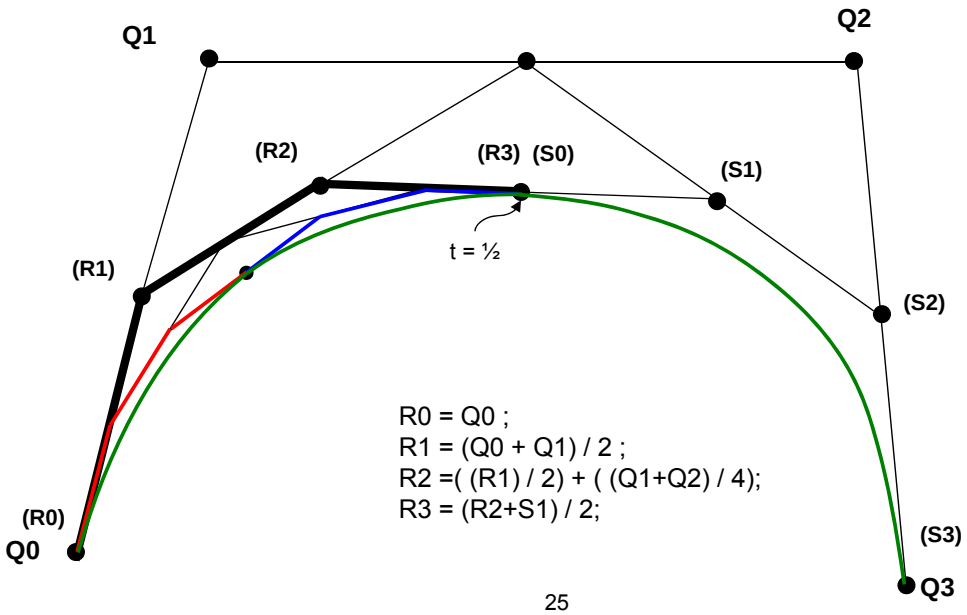
```
/** A routine to draw the (cubic) Bezier Curve represented by the (1x4) input
 *  Control Point Array using iterative plotting along the curve and an explicit
 *  computation which produces a weighted sum of control points for each new point.
 *  Note: This is (Java-like) pseudo code, not real Java code.
 */
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]



Recursive Subdivision

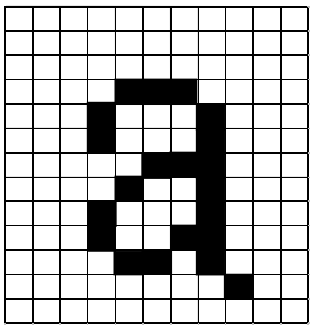
```
/** Draws the (cubic) Bezier curve represented by the (1x4) input Control Point Vector
 *  by recursively subdividing the Control Point Vector until the control points are
 *  within some tolerance of being colinear, at which time the Control Points are deemed
 *  "close enough" to the curve for 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.
*   Note: This is (Java-like) pseudo code, not real Java code. */
void drawBezierCurve (ControlPointVector) {
  if ( straightEnough (ControlPointVector))
      Draw Line from 1st Control Point to last Control Point ;
  else  {
      subdivideCurve (ControlPointVector, LeftSubVector, RightSubVector) ;
      drawBezierCurve (LeftSubVector) ;
      drawBezierCurve (RightSubVector) ;
  }
}
 /** 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) ;
}
```

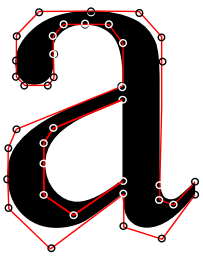
Recursive Subdivision (cont.)

```
/** 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 ;
}
```


Applications Of Curves

- Two types of "fonts"
  - Bit-mapped
  - Outline





a

a

17 - Threads

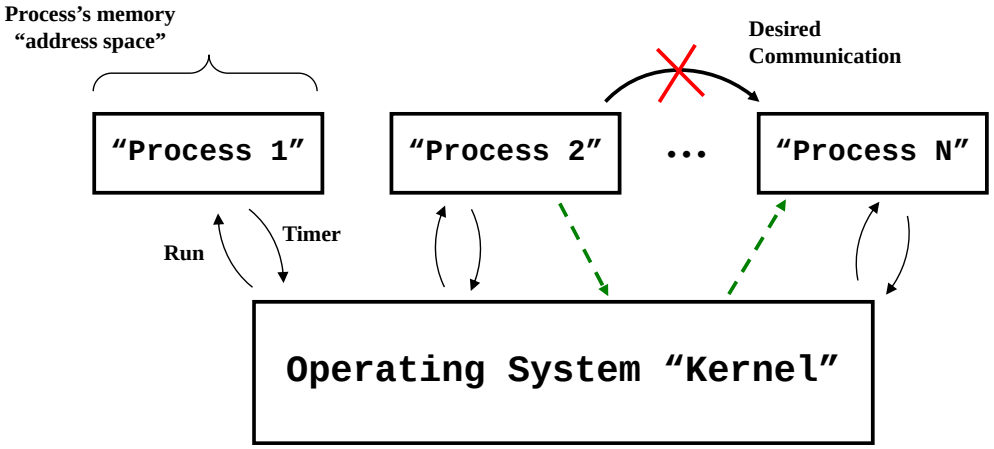
Computer Science Department
California State University, Sacramento

Overview

- **Threads vs. Processes**
- **Java/CN1 Threads**
  - **Thread Class**
  - **Runnable Interface**
- **Thread Synchronization**
- **Application Uses**

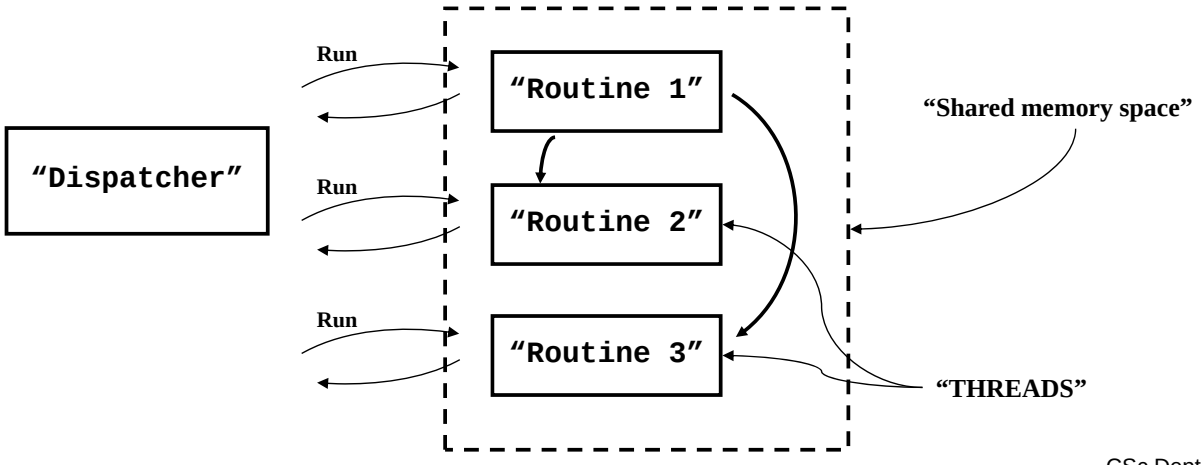
Threads vs. Processes

- 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/CN1 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 CounterTest() {

    // 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;
  private Random rand;

  public Counter(int loopLimit) {
    this.loopLimit = loopLimit;
    rand = new Random();
  }

  // Specify the runnable (thread) behavior.
  public void run() {
    for (int i=1; i<=loopLimit; i++) {
      System.out.println(i);       // display current loop count
      pause(rand.nextFloat());     // 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");
    }
  }
}
```


Example 2: Concurrent Output

```
public class ConcurrentOutput {

  public ConcurrentOutput() {

    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(rand.nextFloat());
    }
    System.out.println ("Counter: done.");
  }
  private void pause(double seconds) {
    ... as before
  }
}
```


Example 3: Multiple User Threads

```
public class MultipleCounters {
  public MultipleCounters() {
    /* 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();
  }
}
```

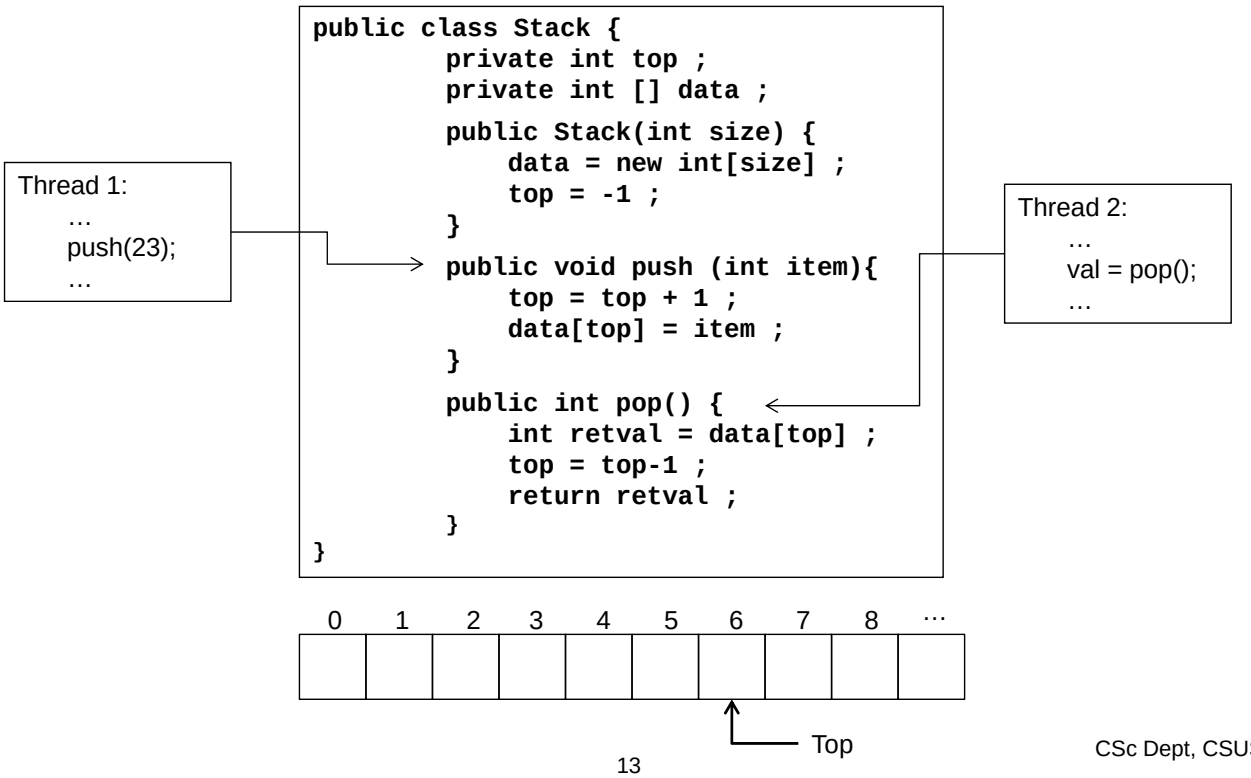
Multiple User Threads (cont.)

```
public class Counter2 implements Runnable {
  private static int totalCounters = 0;    //counts instances
  private int myNum, loopLimit;
  private Random rand;
  public Counter2(int loopLimit) {
    this.loopLimit = loopLimit;
    myNum = totalCounters++;   //assign this instance a unique number
    rand = new Random();
  }
  public void run() {
    for(int i=1; i<=loopLimit; i++) {
      System.out.println("Counter " + myNum + ": " + i);
      pause(rand.nextFloat());
    }
  }
  private void pause(double seconds) { ... }       // as before
}
```


Thread Synchronization

- **Parallel execution can lead to problems**
  - **Corruption of shared data**
  - **Race conditions**
  - **Deadlock**
  - **Starvation**

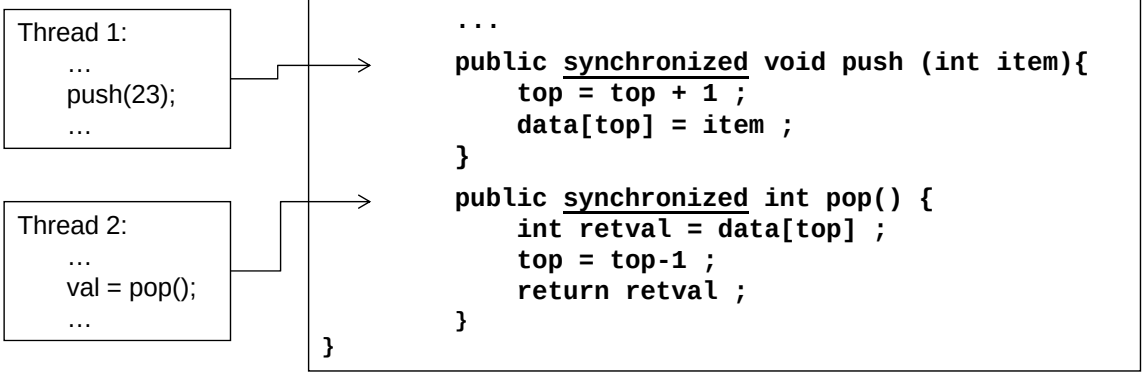
Example: Data Corruption




```
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 ;
        }
}
```


Java/CN1 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 ;
        }
}
```

```
//Another example: synchronized code blocks
public class SomeClass {
   public void someMethod() {
      ...
       synchronized (this) { // a block of code synchronized on this object, note
                             // that instead of "this" you can provide an
                             // arbitrary 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() blocks THIS thread until myOtherThread dies (finishes). Hence, it forces a "sync point" where the threads 'join together'.

Common Thread Uses

- Update vs. Display of Game Worlds
- Event Handling
- Image Loading
- Audio File Playing

18 – Code Signing and Distribution

Computer Science Department
California State University, Sacramento

Overview

- Common Terms in Signing and Distribution
  - Certificate, Provisioning Profile, Development, Distribution, Signing Authority, UDID
- iOS vs Android Signing and Distribution
- iOS/Android Signing and Distribution Details
- Building/Downloading/Installing Native Apps using CN1

Common Terms

- Code Signing and Distribution requires:
  - **Certificate**
  - **Provisioning profile**
- **Development** (debug) is testing your app on your own device whereas **distribution** (release) involves publishing it on a marketplace (e.g., Apple's App Store or Google's Play Store)
- Certificates are issued by a **signing authority**
  - A body that certifies that you are who you say you are

Certificate and Provisioning Profile

- Certificate is like a company stamp or your signature
  - You use it to sign an app so the users know who it is from
  - You can use the same certificate for all your apps
- Provisioning profile gives hints/guidelines for the application installation
  - You choose which devices can run your app and which app services your app can access
  - Gives details about the application and who is allowed to execute it

UDID

- UDID (Universal Device Identifier) identifies mobile devices uniquely
- It is used in creating provisioning profiles
- Some operating systems (e.g., iOS) block access to this value due to privacy concerns:
  - Don't use an iOS app to get the UDID since most return the wrong value. The official way to get the UDID is through iTunes.

Signing Authority

- Apple issues certificates for iOS development
  - To generate a certificate you need to be registered to Apple Developer Program ($99/ year).
  - But our department can register you to this program for free! Let me know if you would like to get registered.
- Android uses self-signed certificates
  - Anyone can ship Android app
  - Certificate indicates that you are the same person who previously ship the app because the app can only be updated with the exact same certificate

Signing and Distribution: iOS vs Android

- iOS process is complicated compared to Android process
- For iOS, you need a separate certificate (.p12 file) and provisioning profile (.mobileprofile file) files
  - For Android, you only need a certificate file (.ks file)
- For iOS, you need certificate and provisioning file for development as well as distribution
  - For Android, you only need it for distribution

Signing and Distribution: iOS vs Android (cont)

- You should re-use the same certificate for all iOS apps you develop
  - This is recommended for all platforms for simplicity
  - But some Android developers prefer creating per-application certificates. This way they can easily transfer ownership of different apps to different developers.
- You need to be registered to the Apple Developer Program for iOS.

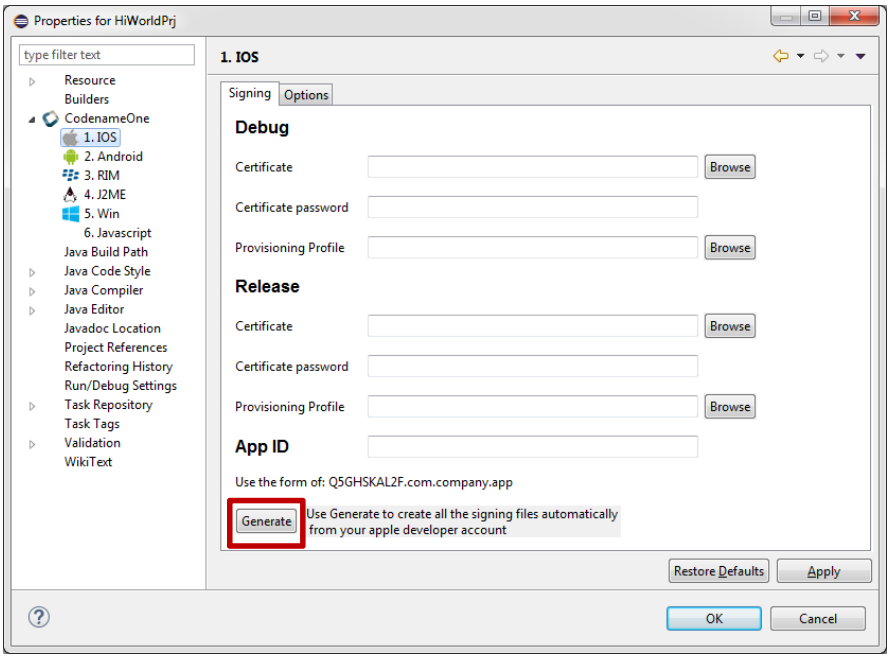
iOS Signing and Distribution Details

- You need a different certificate and provisioning profile files in development and distribution
  - You need to generate two pairs of files (four in total).
- Usually requires a Mac, but:
  - In CN1, You can use iOS Signing Wizard to generate certificates/provisioning files without requiring a Mac or deep understanding of the signing process for iOS
  - You can use other software running on Windows…

iOS Signing Wizard

You can reach the iOS Signing Wizard from:

Project Properties → CN1 → iOS → "Generate" button



iOS Signing and Distribution Details (cont.)

- If you already have the certificate and provisioning profile, you do not need to use the Wizard. Simply enter them to iOS signing properties panel.
- See “Advanced iOS signing” of CN1 Developer Guide for generating certificate/provisioning profile without the Wizard.
- While creating your provisioning file, you need to list UDID of the iOS devices used during development/testing.

iOS Signing and Distribution Details (cont.)

- Apps signed with development certificate can only be installed on one of the iOS devices added to the provisioning profile (e.g., your own device).
- Apps signed with distribution certificate cannot be directly installed to your own iOS device. First, you need to upload it to iTunes. Then you can test this build and submit it to App Store.

Android Signing and Distribution Details

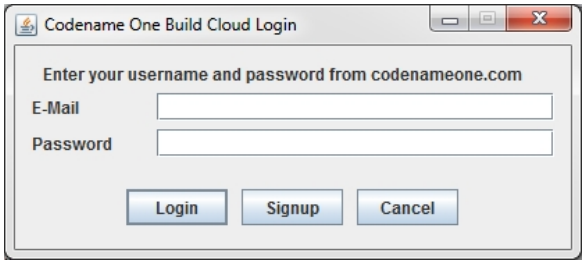
- CN1 provides a wizard also for generating Android certificate (i.e., keystore file with extension .ks)
- The certificate can also be generated manually using JDK's keytool executable
  - See CN1 Developer Guide for more details

Building Native Apps in CN1

- After you populate the signing properties panel for iOS/Android with proper certificates (and provisioning profiles), you can send your app to CN1 build servers to generate native apps.
- Build servers take in CN1 code and generate a native code that can run on Android/iOS/Windows Phones, Mac/Windows Desktop environments, etc...
- Then you can upload these native apps on your devices (in development mode) or distribute them on marketplaces (in distribution mode).

Building Native Apps in CN1 (cont.)

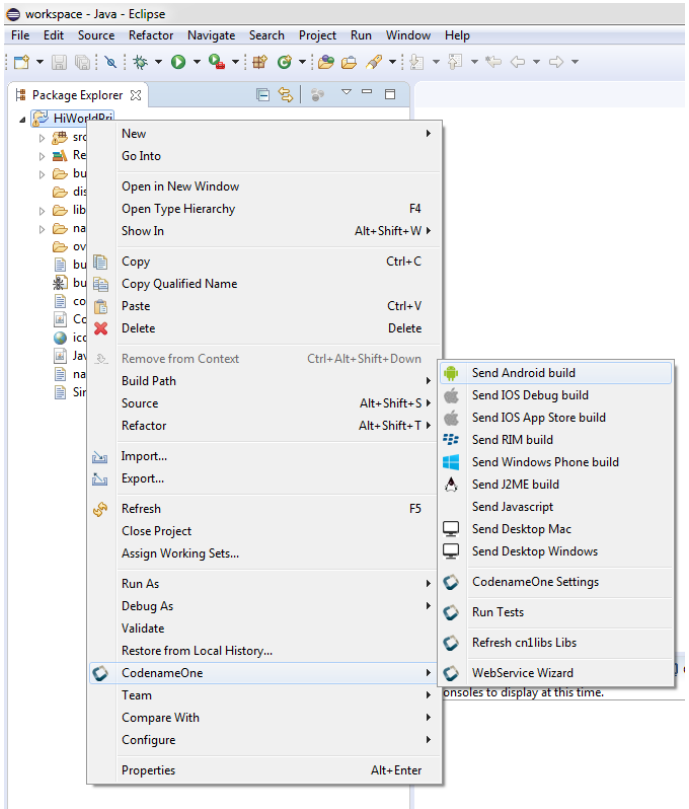
- First you need to create a Codename One user id and password.
- You need to enter this id/password first time you send your code to build servers:



- Send your code to servers as follows:

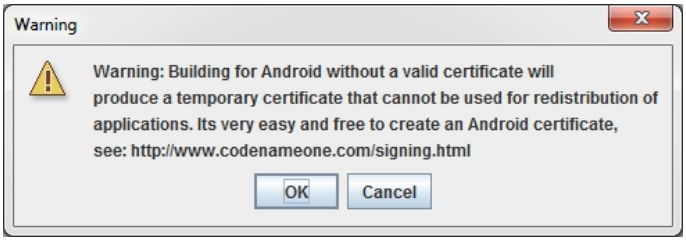
Right Click on Project → Codename One → Send to Android Build (or Send to iOS Debug/App Store, etc…)

Sending CN1 Code for Android Build



Sending CN1 Code for Android Build (cont.)

- If you have NOT specified a certificate for your project, you will receive this warning:

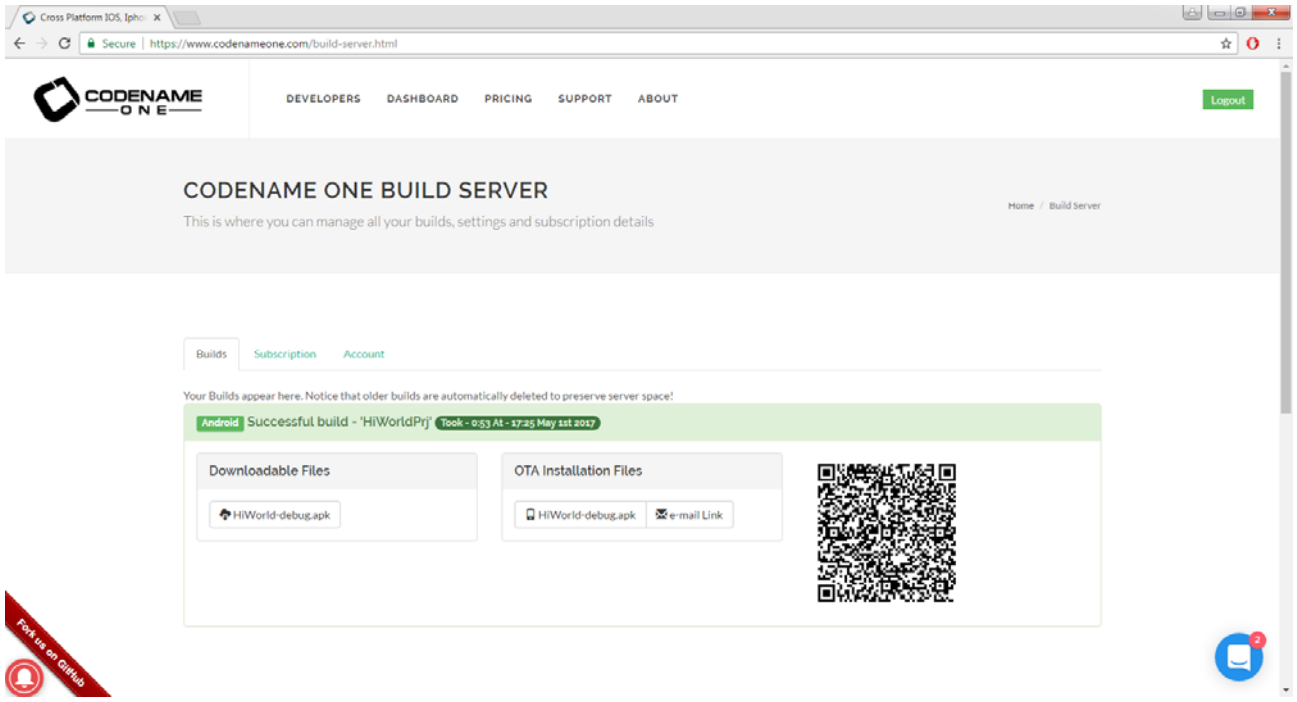


- This means that after you download the build to your Android device, to install it, you need to change the device settings to allow installation from “Unknown Sources”…

Downloading/Installing Native Apps

- Sign in with the same id/password to the CN1 website (codenameone.com).
- Go to “Dashboard” to see the status of your build.
- When the build is done, download and install it to your device (i.e., the .apk file for Android app).
- You can use a QR reader to download the development version of your app to your device.

Downloading the Native Android App



Appendices

California State University, Sacramento
Department of Computer Science

Java Basics

- 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 Windows command line:

  set PATH=C:\Program Files\Java\jdk1.8.0_144\bin;%PATH%

- Path to the current directory (indicated by a single period) must be included in the "CLASSPATH" variable; e.g. in Windows command line:

  set CLASSPATH=.;%CLASSPATH%

- Path to the Java home directory must be defined as the "JAVA_HOME" variable; e.g. in Windows command line:

  set JAVA_HOME= C:\Program Files\Java\jdk1.8.0_144

  Note that "set" commands mentioned above set the variables temporarily (will only be valid for processes that will be launched from the command window that we have typed the "set" command). **To set them permanently in Windows, go to "Control Panel -> System -> Advance System Settings -> Environment Variables (add it to "System Variables")".**

Java Built-in Primitives (8 kinds):

INTEGER types:

- **byte** (-128 .. +127)
- **short** (2 bytes: -32768 .. +32767)
- **int** (4 bytes: -2G .. +2G)
- **long** (8 bytes: $\pm 2^{63}$)

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:

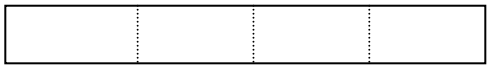
(example from Jia: OO Software Development, A-W, 2000)

```
/** 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) ;
      }
}
```

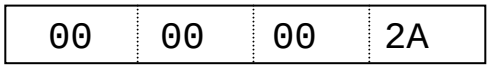
REFERENCES:

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

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

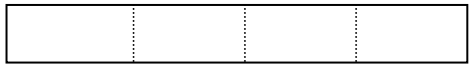


j = 42 ; //assignment stores a value in the space:



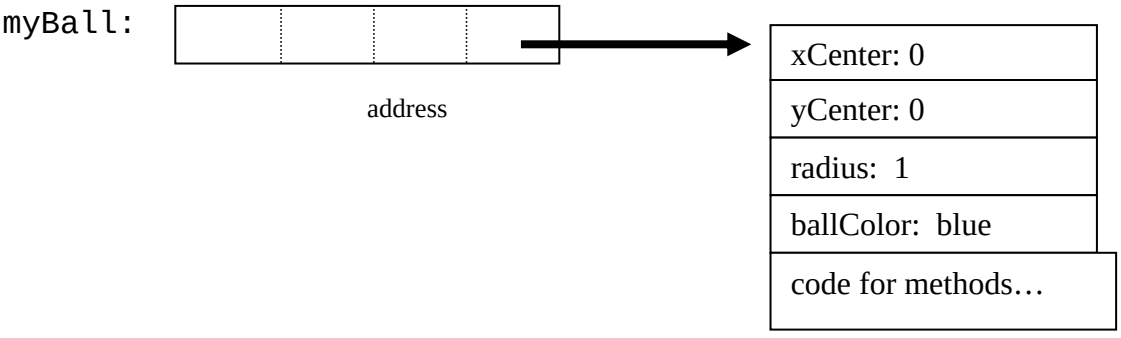
- A variable representing an object holds the address of the object, called a *reference*:

```
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;
```

- 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 "equals**I**gnoreCase()"

- A common mistake:

```
...
if (s1 == "Ed") {
     System.out.println ("The string contains 'Ed' ");
}
```

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

- Correct approach:

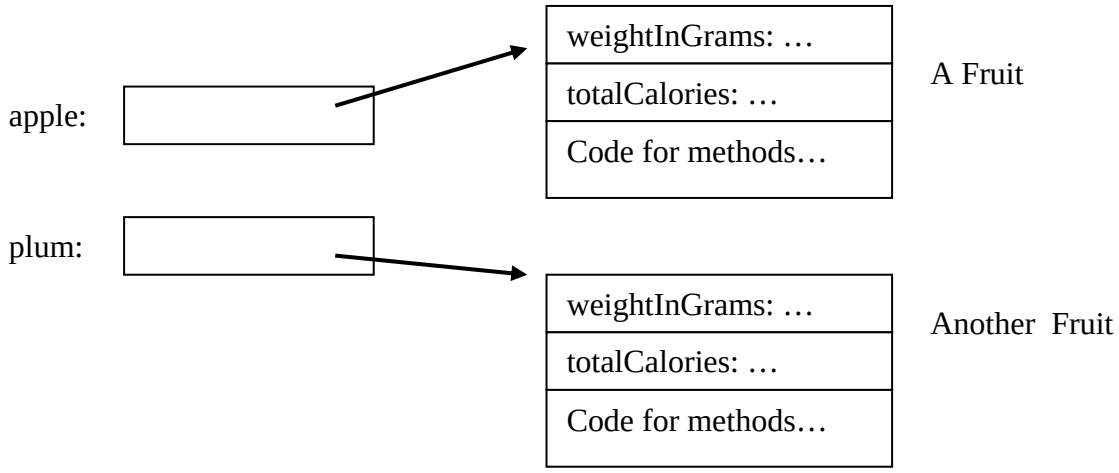
```
if (s1.equals("Ed"))  { ... }
```

GARBAGE COLLECTION:

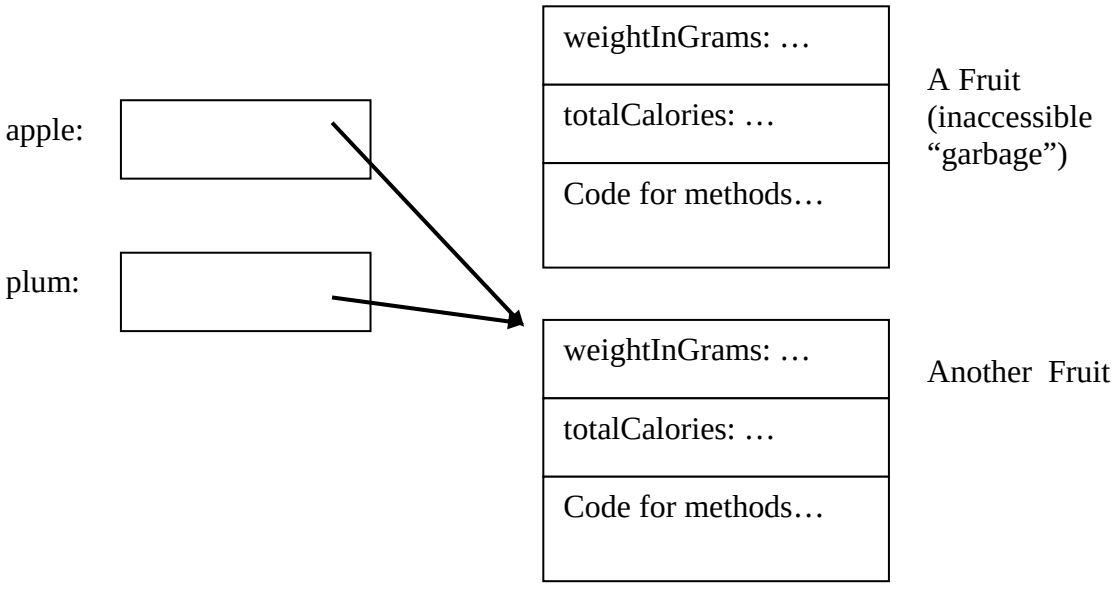
- Important characteristic: assignment does not copy objects; it copies references:

```
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:

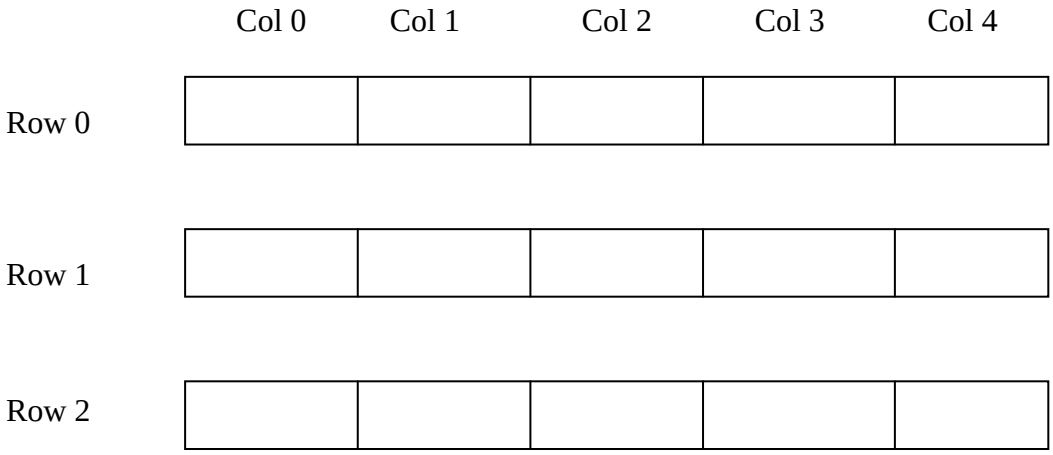
Declaration :

```
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:



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) ;
     }
}
```

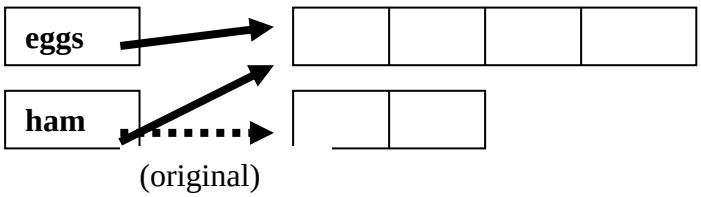
ARRAY ASSIGNMENT:

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):



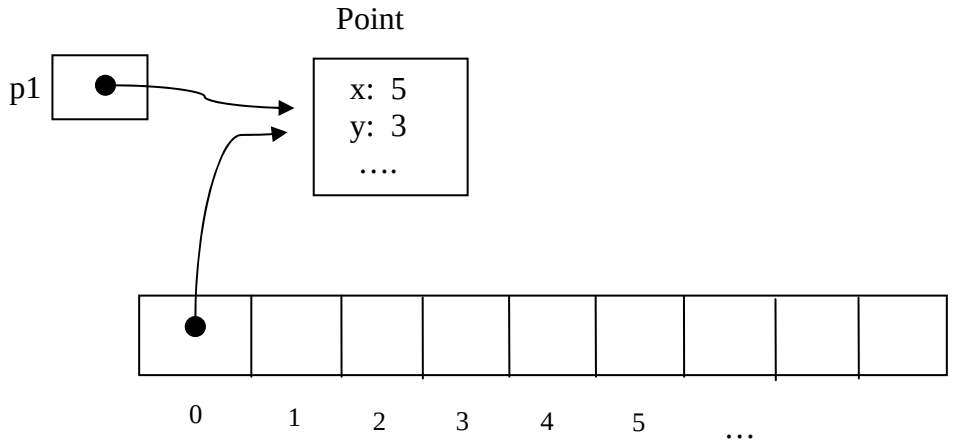
(original)

- 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?

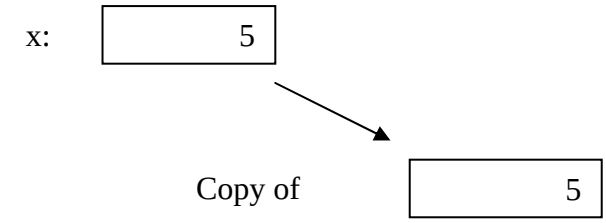
- Answer: **`https://docs.oracle.com/javase/8/docs/api/`**
    - Complete online Java Standard Edition 8 API documentation
    - Can be downloaded to your machine

PARAMETERS:

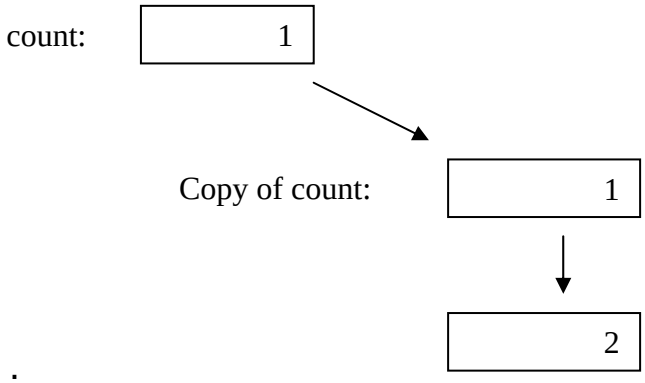
- ALL parameters are passed using "Call by Value" – NEVER "Call by Reference"
  - A *copy* of the actual value of the parameter is passed
- The original parameter CANNOT be modified by any method to which it is passed
- The *effect* of this *appears* to be different for **primitives** and **objects**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

California State University, Sacramento
Department of Computer Science

Elements of Matrix Algebra

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:

	0	1	2
0	X	X	X
1	X	X	X
2	X	X	X
3	X	X	X

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:

$$2 * \underbrace{\begin{pmatrix} 1 & 5 \\ -2 & 4 \\ 3 & 1 \end{pmatrix}}_{M1} = \underbrace{\begin{pmatrix} 2 & 10 \\ -4 & 8 \\ 6 & 2 \end{pmatrix}}_{M2}$$

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:

$$\underset{A}{\begin{pmatrix} 1 & 5 \\ -2 & 4 \\ 3 & 1 \end{pmatrix}} + \underset{B}{\begin{pmatrix} 2 & 2 \\ -4 & 6 \\ -3 & 1 \end{pmatrix}} = \underset{C}{\begin{pmatrix} 3 & 7 \\ -6 & 10 \\ 0 & 2 \end{pmatrix}}$$

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):

$$\begin{bmatrix} 2 & 3 \end{bmatrix} \qquad \begin{bmatrix} 4 & 6 & -1 \end{bmatrix}$$

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>.

$$\underset{V1}{\begin{bmatrix} 2 & 3 \end{bmatrix}} \quad * \quad \underset{M1}{\begin{pmatrix} 1 & 2 & -1 \\ 4 & 5 & 0 \end{pmatrix}} \quad = \quad \underset{V2}{\begin{bmatrix} 14 & 19 & -2 \end{bmatrix}}$$

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) :

$[2\ \ 3] \bullet [1\ \ 4] = (2*1) + (3*4) = 14$;

$[2\ \ 3] \bullet [2\ \ 5] = (2*2) + (3*5) = 19$; and

$[2\ \ 3] \bullet [-1\ \ 0] = (2*-1) + (3*0) = -2.$

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:

$$\begin{pmatrix} 2 \\ 3 \end{pmatrix} \qquad \text{or} \qquad \begin{pmatrix} 4 \\ 6 \\ -1 \end{pmatrix}$$

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).

$$V2 = \begin{pmatrix} 14 \\ 19 \\ -2 \end{pmatrix} \quad = \quad M1 = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ -1 & 0 \end{pmatrix} \quad * \quad V1 = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

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).

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 $A^{\mathbf{T}}$.

$$A = \begin{pmatrix} 2 & 10 & -3 \\ -4 & 8 & 4 \\ 6 & 2 & -7 \end{pmatrix} \qquad A^{T} = \begin{pmatrix} 2 & -4 & 6 \\ 10 & 8 & 2 \\ -3 & 4 & -7 \end{pmatrix}$$

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:

$$B = \begin{pmatrix} 2 & 10 \\ -4 & 8 \\ 6 & 2 \end{pmatrix} \qquad B^{T} = \begin{pmatrix} 2 & -4 & 6 \\ 10 & 8 & 2 \end{pmatrix}$$

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. $(A + B)^{T} = A^{T} + B^{T}$; that is, the transpose of a sum of two matrices equals the sum of the transposes of the individual matrices.

2. $(A * B)^{T} = B^{T} * A^{T}$; 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:

$$\underset{A}{\begin{pmatrix} 2 & 3 & -1 \\ 4 & 0 & 6 \end{pmatrix}} * \underset{B}{\begin{pmatrix} 1 & -1 & 0 & 4 \\ 2 & -2 & 1 & 3 \\ 5 & 7 & 1 & -3 \end{pmatrix}} = \underset{C}{\begin{pmatrix} 3 & -15 & 2 & 20 \\ 34 & 38 & 6 & -2 \end{pmatrix}}$$

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 is *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:

$$\underset{P2}{\begin{pmatrix} X' \\ Y' \end{pmatrix}} = \underset{M}{\begin{pmatrix} & \\ & \end{pmatrix}} * \underset{P1}{\begin{pmatrix} X \\ Y \end{pmatrix}}$$

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.

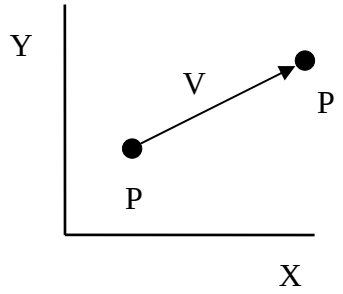
California State University, Sacramento
Department of Computer Science

Elements of Vector Algebra

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:



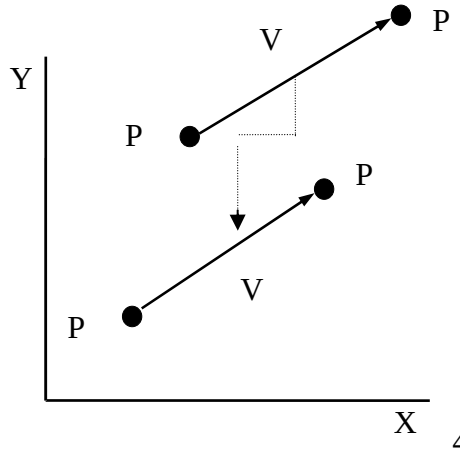
$V = (P2 - P1)$

$= [\ (X_2-X_1)\ \ (Y_2-Y_1)\ ]$

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*):



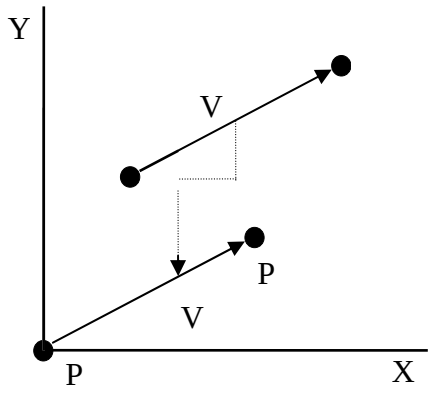
V1 = (P2 - P1)

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:



$V = (P2 - P1)$

$= [\ (X_2 - X_1)\ \ (Y_2 - Y_1)\ ]$

$= [\ (X_2 - 0)\ \ (Y_2 - 0)\ ]$

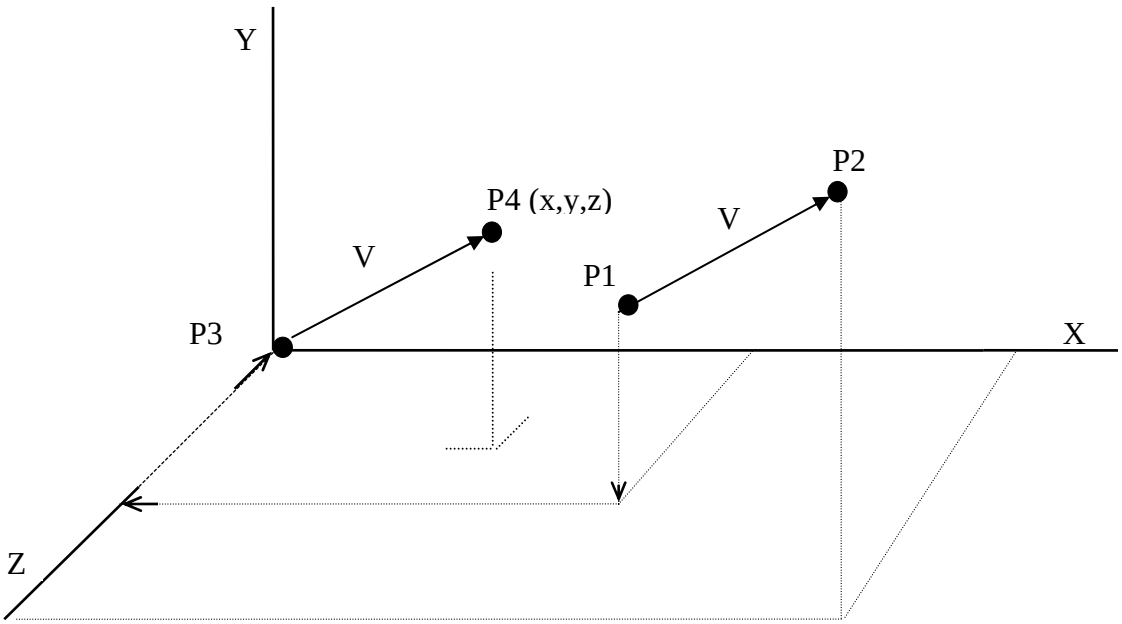
$= [\ (X2)\ \ (Y2)\ ]$

$= [\ X2\ \ Y2]$

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,

$V = (P2 - P1)$

$= (P4 - P3)$

$= [(X_2 - X_1)\ (Y_2 - Y_1)\ (Z_2 - Z_1)]$

$= [(X_4 - X_3)\ (Y_4 - Y_3)\ (Z_4 - Z_3)]$

$= [(X_4 - 0)\ (Y_4 - 0)\ (Z_4 - 0)]$

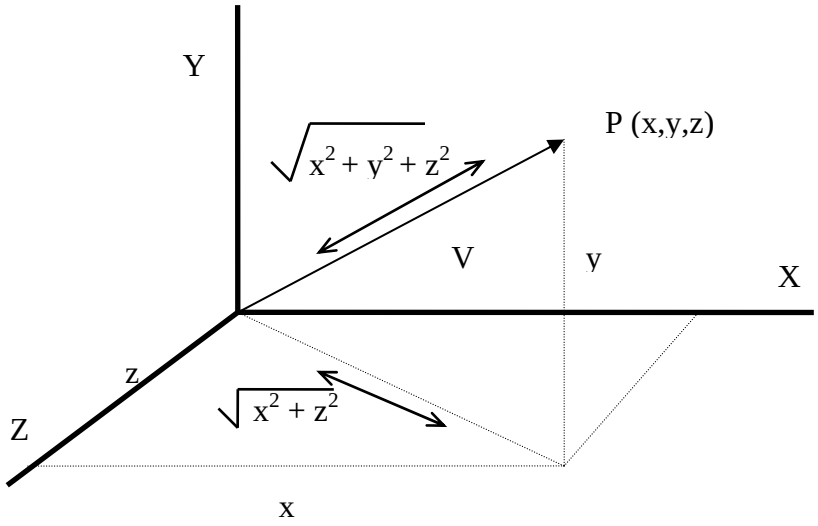
$= [X_4\ Y_4\ Z_4]$

$= [x\ y\ z]$

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:

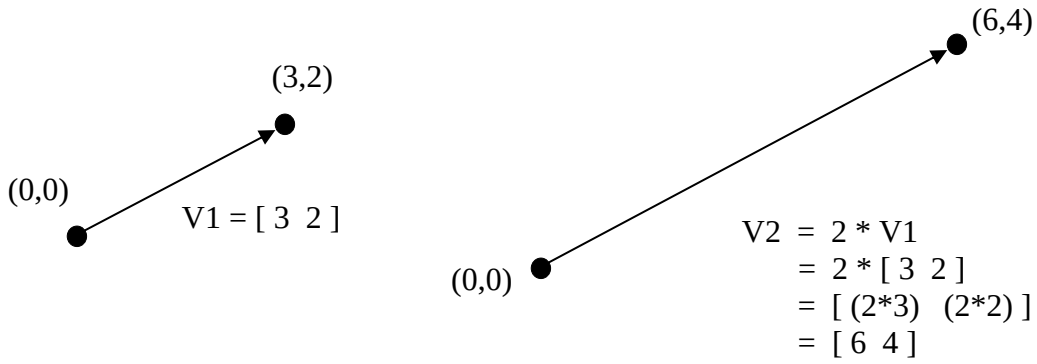


The magnitude (length) of V is $|V| = \sqrt{(x^2 + y^2 + z^2)}$.

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 $|V| =$ sqrt $(x^2 + y^2)$.

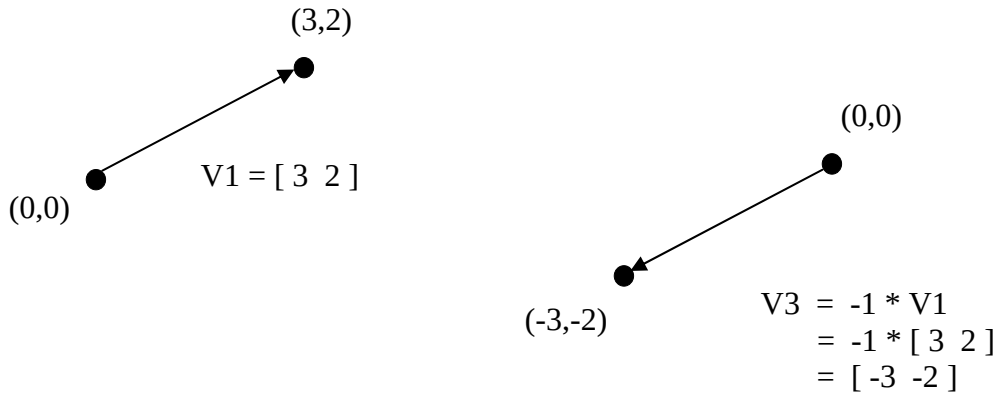
5. Scalar Multiplication

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

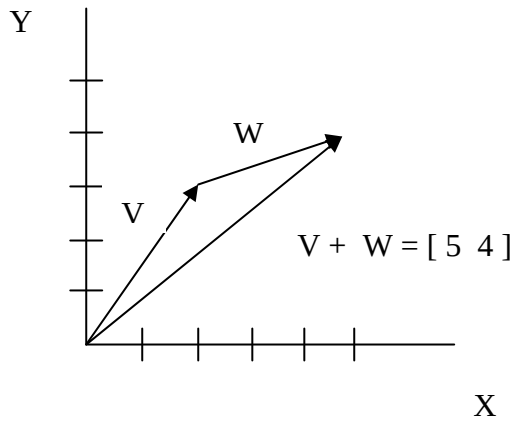
Given a vector V, there exists a corresponding vector U (also sometimes denoted as V with a "^" over it) called a *unit vector* with the property than U has the same *direction* as V but has *length = 1.*

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 $|V| = \text{sqrt}(3^2 + 4^2) = 5$, 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 $|U| = \text{sqrt} (0.6^2 + 0.8^2) = \text{sqrt} (0.36 + 0.64) = \text{sqrt} (1.0) = 1.0$.

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

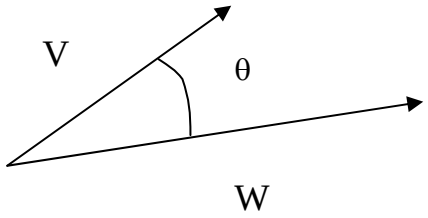
Another well-defined operation on vectors is called the *dot product* (also called the *inner product).* Given two vectors V and W, the dot product is written notationally as V • W and is defined as a scalar value $D = \Sigma (V_i * W_i)$ over all components "i" of V and W. Note that the dot product 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

$$\cos(\theta) = (V \bullet W)$$