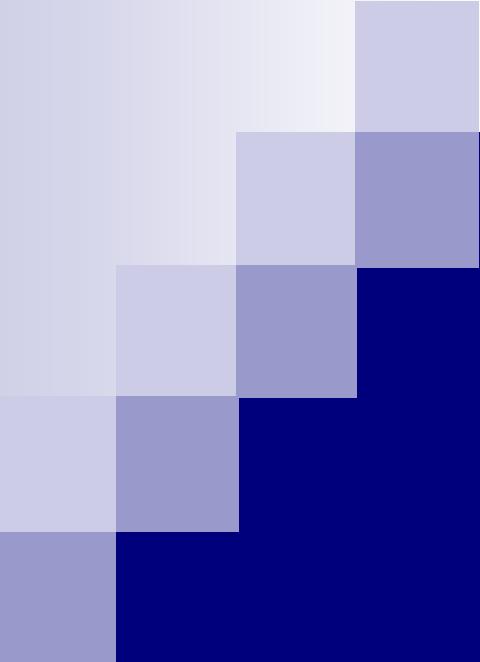


CSc 133 Lecture Notes

1 – Course Introduction

Computer Science Department
California State University, Sacramento



CSC 133

Agenda

- Course Introduction
- Introduction to Codename One (CN1) and Assignment #0
- Student attendance and introduction
- Adding requests

Overview

- Instructor Information
- Classroom conduct
- Prerequisites
- Course topics
- Texts and references
- Grading: exams and programs
- Communication
- Workload
- Ethics
- Grading scheme for the course assignments

Contacting Your Instructor

Dr. Doan Nguyen

Office: Riverside Hall 5009

Phone: (916) 278-6834

Office Hours: M 9:00 AM –12:00 PM
and by appointment

Email: doan.nguyen@csus.edu

Webpage: <http://ecs.csus.edu/~nguyendh>

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!

- o Language implementation:
 - Abstraction
 - Encapsulation
 - Inheritance
 - Polymorphism
- o Tools supporting OOA/OOD/OOP:
 - formalisms such as *UML*
 - ***design patterns (underlying theme of CSC 133!)***

Second topic: Computer Graphics

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

Mobile App Development

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 (Available weekly), available at the “Files” section of LMS (Canvas)
 - Credits: Dr. Muyan-Ozcelik Pinar
 - Codename One Developer Guide:
CN1 Developer Guide - Revision 4.0 (May 5, 18)
 - 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 materials:

- Basic Debugging With Eclipse:
<https://www.youtube.com/watch?v=PJWtO5wrptg>
- UML 2.0 Class Diagram:
<https://www.youtube.com/watch?v=3cmzqZzwNDM>

Grading

- Weighted Curve based on:
 - Programming Assignments (4) 40%
 - Midterm Exam 20%
 - Final Exam 25%
 - Quizzes 10%
 - Roll Attendance 5%
- Additional Criteria
 - Not to miss more than 1 week of class
 - Passing completion of :
 - Programming assignments (combined)
 - Exams (Midterm + Final combined)
 - Attendance/Quizzes

Grading (cont.)

Programming Assignments

- Required to be solved using CN1, submitted via LMS (Canvas)
- 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 1 week** 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)
- Must keep a *backup* (machine-readable) copy

Grading (cont.)

Exams

- o Dates are noted on the outline
- o Final Exam as scheduled by University
- o Study Guides will be provided
 - Only one sample exam will be provided prior midterm exam but not the final exam.
 - *Only the course notes are complete!*
- o Make-up exams only under extreme circumstances:
 - *generally requires prior arrangements*

Grading (cont.)

Attendance Quizzes

- o Usually held at the end of lectures
- o There is no makeup quiz allowed

Computers

- Work on any school machine or your machine which have CN1
- To install CN1:
 - Install version of Java SE JDK 1.8
 - Install latest version of Eclipse for Java Developers
 - Install CN1 as a plugin to the Eclipse

(installation instructions will be discussed in class)

Communication

- LMS (Canvas): **canvas.csus.edu**
 - assignments
 - announcements (via LMS (Canvas) with emails)
 - feedback and grades
- Check your SacLink email and LMS (Canvas)
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 LMS (Canvas)

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.

Grading scheme for the course assignments

Concerned Areas	Points/Percent off
Submission File was not following zip file standardizations as mentioned in lecture.	-3 points
Jar file is not runnable from command line invocation.	-10 points
Major functionality was not implemented or not correctly executed.	-5 points for each area (Maximum -15 points)
Minor functionality was not implemented or not correctly executed.	-2 points for each area (Maximum -10 points)
Program raises error and aborted during execution. This required a restart.	-5 points (Maximum -10 points)
Coding Styles: Program documentation (comment), Variable Usage, Hardcoding, Unstructured code, Indentation, etc .	-2 points for each area (Maximum -10 points)
Correct UML Diagram: Class, Attribute, Method, Relationship, usage of proper design patterns	-3 points for each area, with a maximum of -6 points (Maximum -15 points for the entire UML diagram)
Late work	-5 percent for each day after the due date. 0 points after 1 week

CSC 133 resources

(I need help!)

- Visit Instructor Office Hours
- Utilize demonstration source codes (to be supplied along with lecture notes)
 - Need to change package name.

CSC 133 resources (Cnt)

- Canvas Section Discussion
 - Questions regarding assignments
 - Suggested ideas to solve specific issues
 - Please: **NO CODE SHARING PLEASE (Graders will check)**
 - A small portion of points to be awarded to students who contribute to **productive** collaborative discussions.

CSC 133 resources (cnt)

(I need help!)

- Visit Tutor Center: (Old schedule – Fall 2018)

ECS Tutoring Services for Computer Science

Free Services Santa Clara, Room 1217

Open September 10, 2018 – November 30, 2018

Tutoring CLOSED November 19-23, 2018 and November 12, 2018

Last Updated: September 10, 2018

Times:	Monday	Tuesday	Wednesday	Thursday
10:30-11:00	Ricky Gutierrez	Paul McHugh		
11:00-12:00	(10:30-12:00)	(10:30-12:00)	Ricky Gutierrez	
12:00-1:00	Paul McHugh	Oksana Ruderman	(11:00-1:00)	Oksana Ruderman
1:00-1:30	(12:00-1:30)	(12:00-1:30)	Paul McHugh	(12:00-1:30)
1:30-2:00	Nicole Barakat (1:30-4:00)		(1:00-2:30)	
2:00-2:30				
2:30-3:00			Nicole Barakat	
3:00-4:00			(2:30-4:00)	

Questions?

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 (School)

At School Lab's Machines:

- + Eclipse Oxygen.2 Release (4.7.2)
- + Codename One version 3.8.1
- + Java JDK version 1.8.0.191 (current release is 1.8.0.201)

CN1 Installation (@ Student Desktop)

- Install latest version of Java SE JDK (version 8):

<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 CSc-133, “Eclipse IDE for Java Developers – IDE 2018-12” is **recommended**:
<https://www.eclipse.org/downloads/>
- Windows is recommended.



CN1 Installation (cont.)

- Install CN1 plugin 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



Codename One 5.0

Codename One is an open source platform for native mobile development in Java/Kotlin. It natively supports Android, iOS, Windows (UWP) etc. It's free, open... [more info](#)

by [Codename One](#), Other Open Source
[open source](#) [java](#) [iPhone](#) [iPad](#) [iOS](#) ...

1/19/19

- 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 **check “Java 8 project”**). Hit “Next”.
 - Give a main class name “Starter”, 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

CSc Dept, CSUS

CN1 Hello World App

Getting Started (Eclipse)

Eclipse

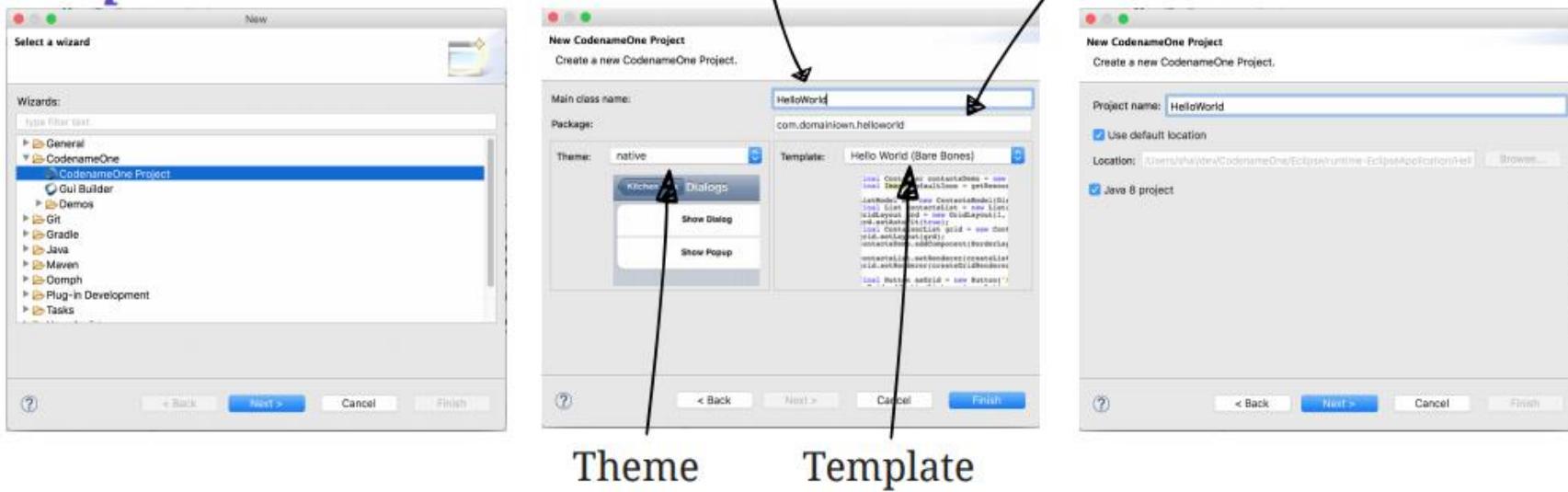


Figure 1. 1. The New App Wizard

Credits: Create an Uber Clone in 7 Days (Shai Almog – Codename One Academy)

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 games, better to use a tablet skin (e.g., iPad, Download it via “Skins” → “More” -> “iPad III iOS 7”)

Troubleshooting Problems

- If dist\HiWorldPrj.jar is not generated:

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_102 (to “System Variables”)

- If the command line complains that:

‘java’ is not recognized ... : add C:\Program Files\Java\jdk1.8.0_102\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

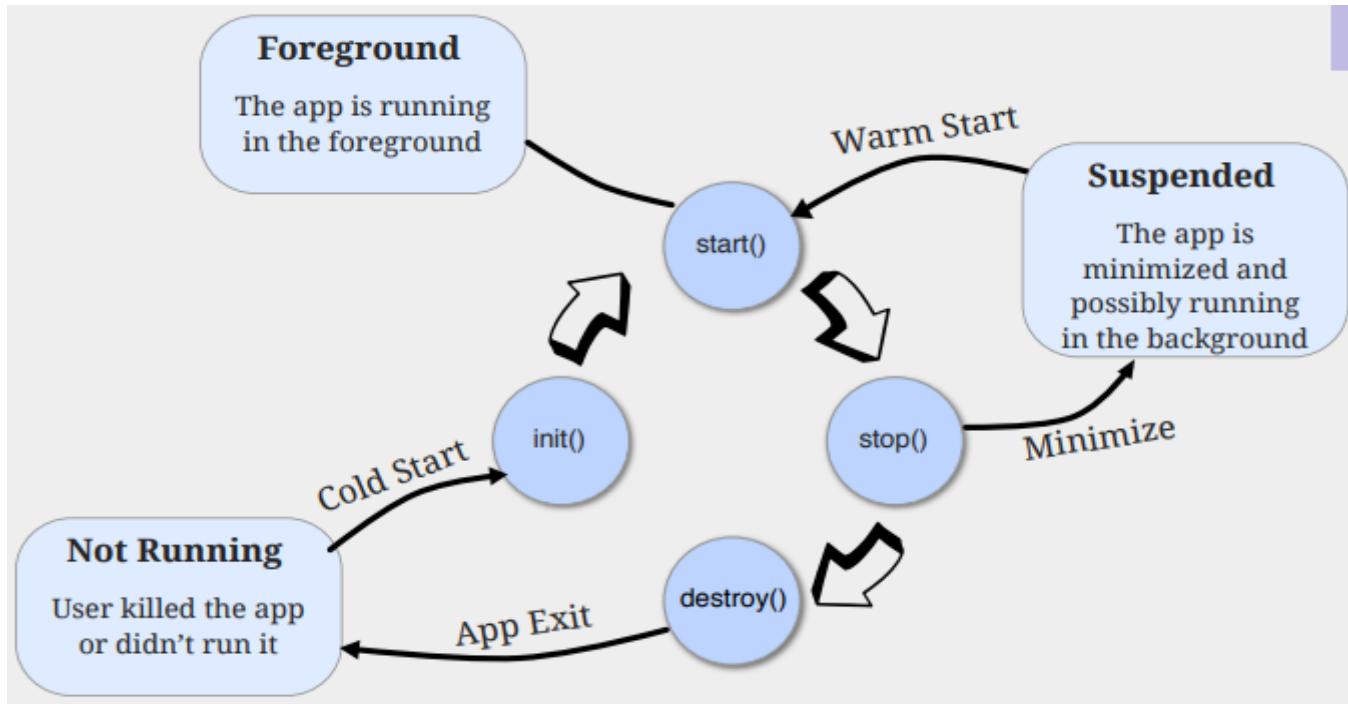
- For each assignment create a different CN1 project.
- You **must** create all assignments in the same way as HiWorldPrj example:
 - (Check “Java 8 project”), select “native” theme, and **“Hello World (Bare Bones)”** template.
 - change the project, main class, and package names...

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(...) {...} // init() is used when the App is launched (NOT resumed!).  
    public void start() {...} // start() is used when application come on foreground (After  
    // init, or after being minimized)  
    public void stop() {...} // stop() is used when application goes to background (ie  
    // minimized)  
    public void destroy() {} // destroy() is used when the OS kill the process  
}
```

CN1 Application Life Cycle



- Foreground – it's running and in the foreground which means the user can physically interact with the app
- Suspended – the app isn't in the foreground, it's either paused or has a background process running
- Not Running – the app was never launched, was killed or crashed

CN1 and Assignments (cont.)

- Solve the assignment by modifying **start()** in Starter.java (**do NOT delete other methods**) and adding/modifying other necessary source files.
- **Make sure** dist\A1Prj.jar is up to date (if not, in Eclipse, right click on dist directory and say “**Refresh**”)
- **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 Canvas.

Grader's Run Program

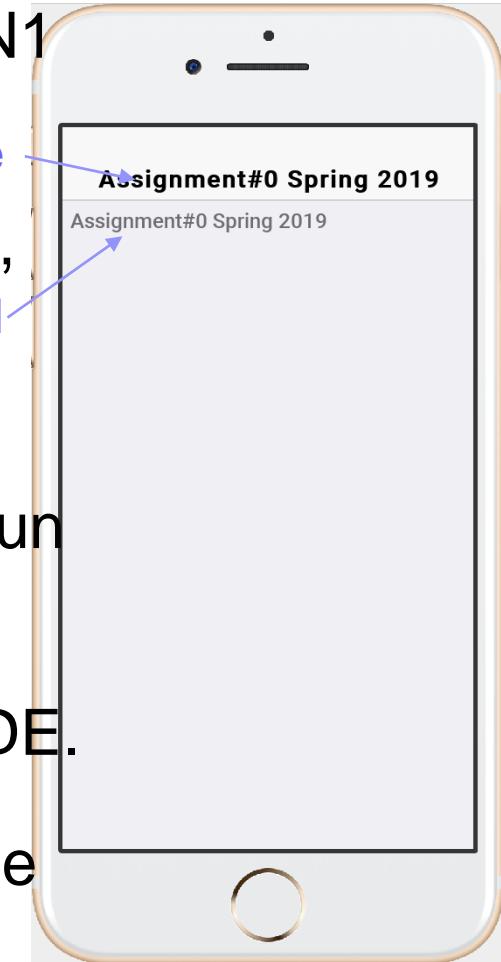
- `java -jar RunAssignment.jar AXPrj.jar`

Where X is replaced with 0, 1, 2, 3, or 4 assignment number.

- Student can also download the same program from Canvas (week 1 folder) and use it.
- Be sure to place RunAssignment.jar at ~~A~~XPrj directory before running.

Assignment#0

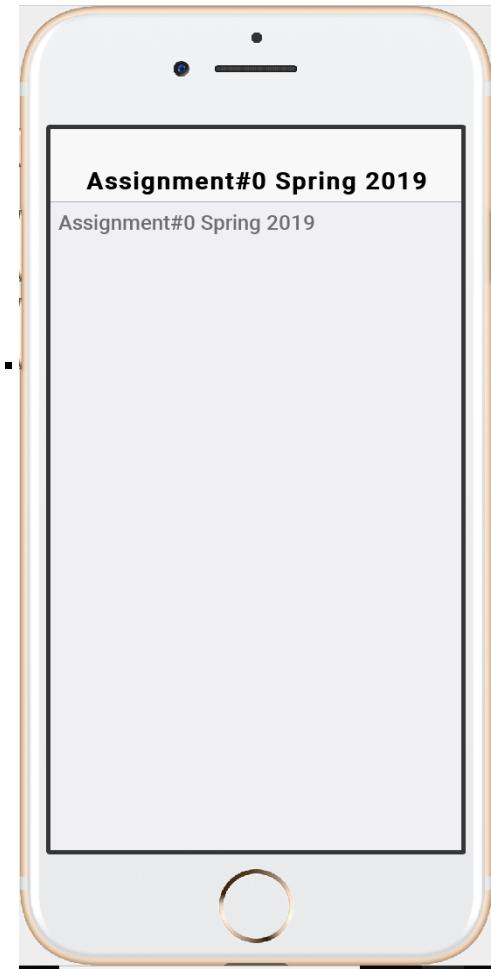
- Find a lab computer that has CN1 or install CN1 to your computer.
- Following the instruction in the previous slides, generate an empty project called A0Prj.
- Modify Starter.java by replacing the texts “Hi World” with “**Assignment #0 Spring 2019**”. Run the simulator.
- Experiments with debugging options of your IDE.
- Verify that your submission also works from the command line.



Do submit A0 via Canvas for Grading (its purpose is to make sure you have access to CN1 and ready to solve real assignments)

Demonstration

- Installing CN1 Plugin.
- Creating a CN1 Project and run a simulation.
- Running the CN1 app from command line.
- Debugging CN1 application.
- Running Grader's program to test.



Other CN1 Online resources

- Developers guide:

CN1 Developer Guide

(<https://www.codenameone.com/files/developer-guide.pdf>)

- 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

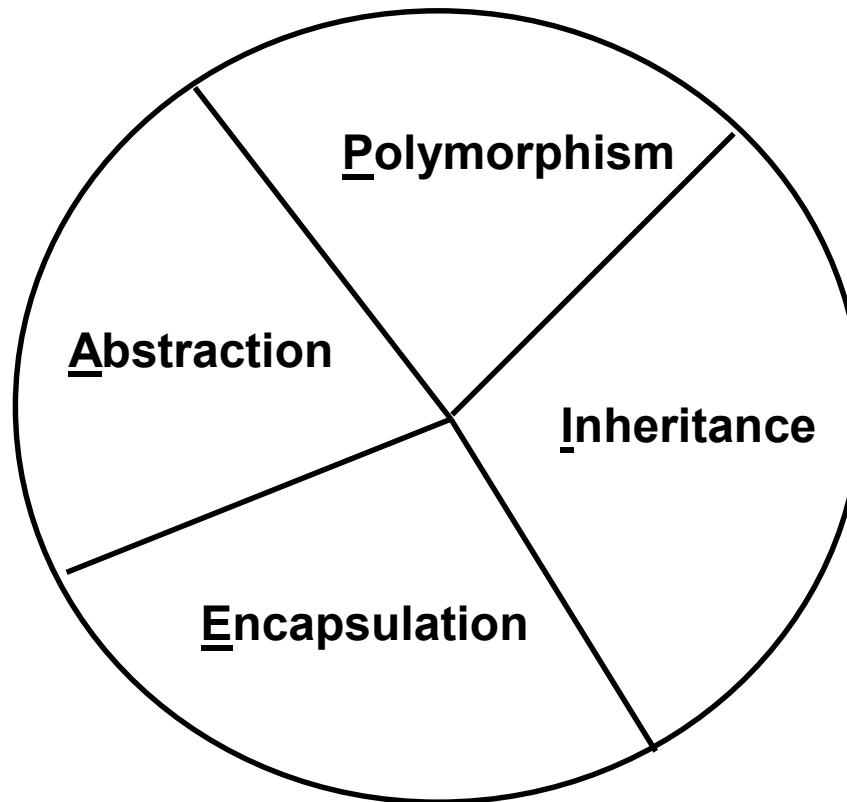
Computer Science Department
California State University, Sacramento

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

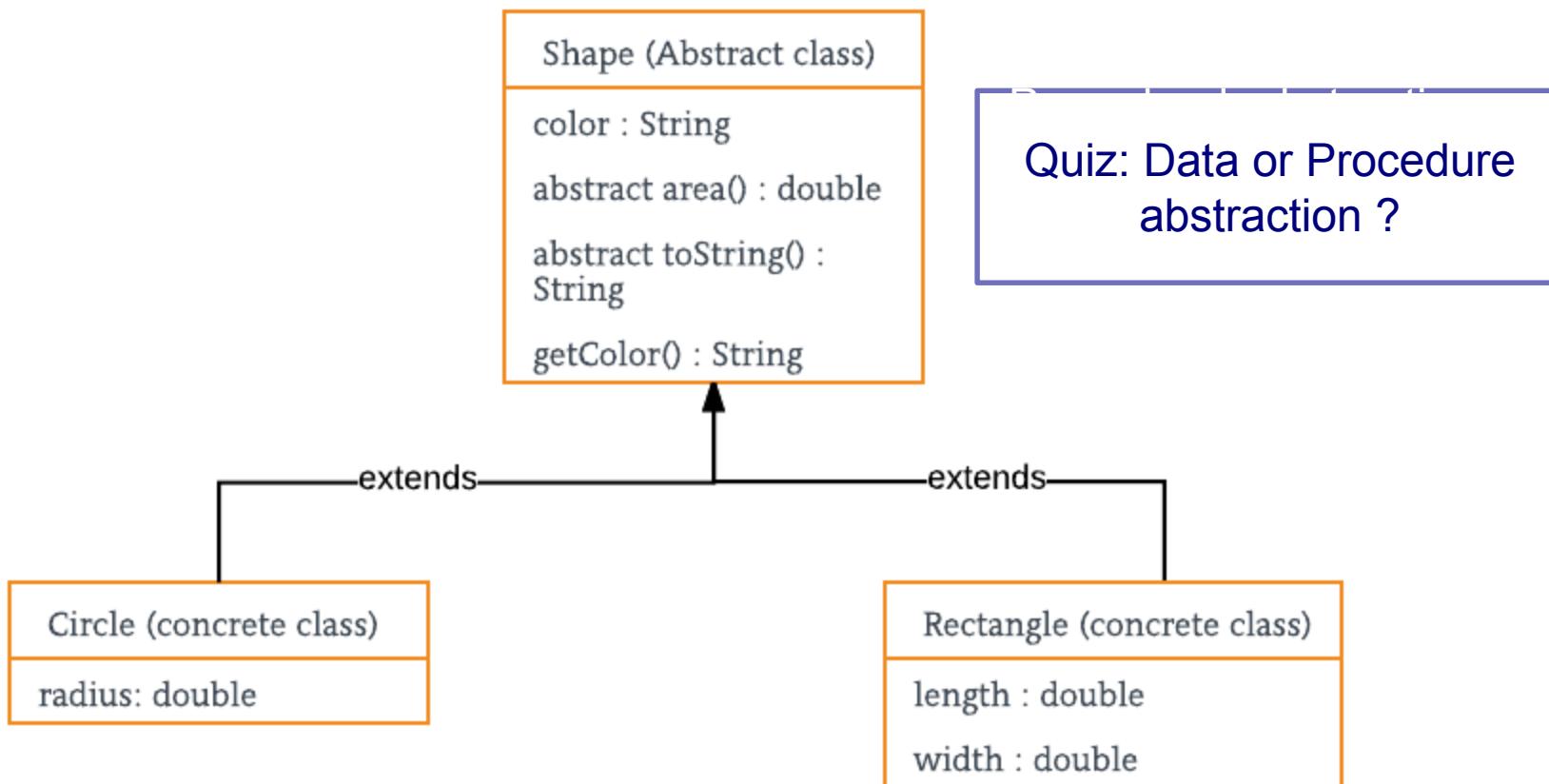
Abstraction is the process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics.

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)

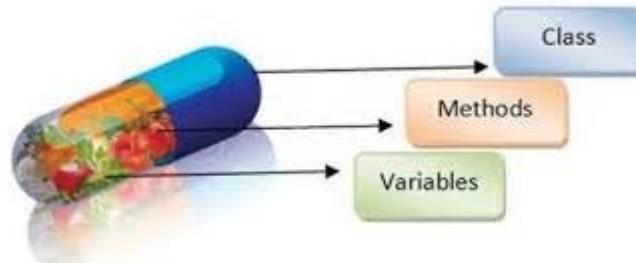
Abstraction Example



The base type is “shape” and each shape has a color, size and so on. From this, specific types of shapes are derived(inherited)-circle, square, triangle and so on – each of which may have additional characteristics and behaviors.

Encapsulation

Encapsulation refers to the bundling of data with the methods that operate on that data.

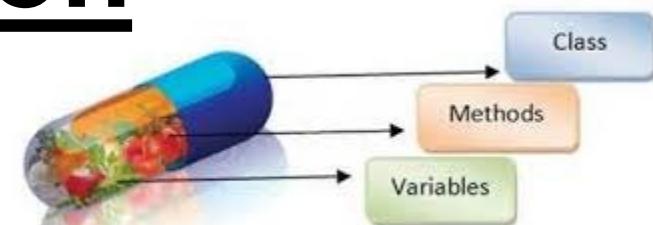


Encapsulation

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

Questions: (1) Name the Constructor ?
(2) Usage ?

Access (Visibility) Modifiers

Java:

Modifier	Access Allowed By			
	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<none>	Y	Y*	N	N
private	Y	N	N	N

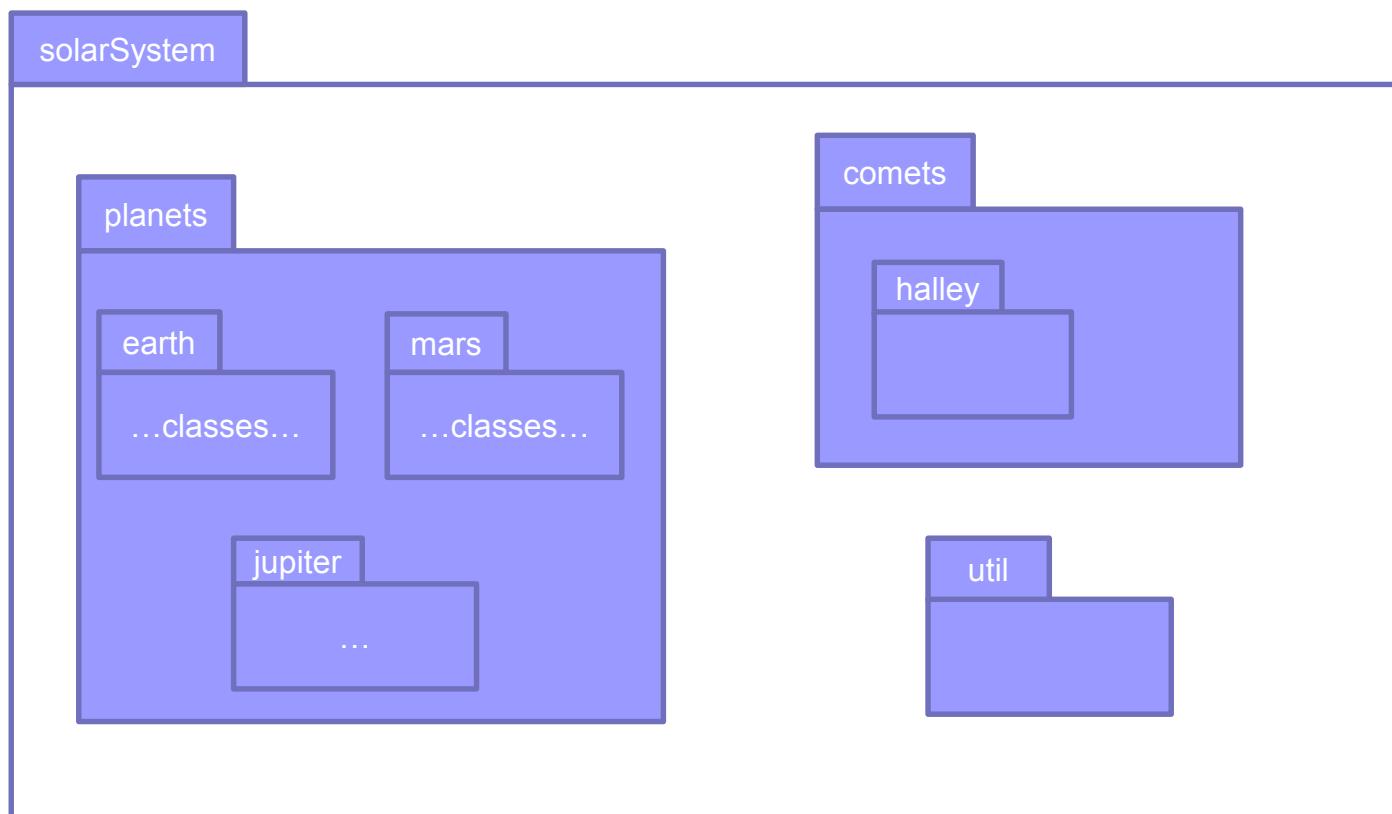
C++:

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*”

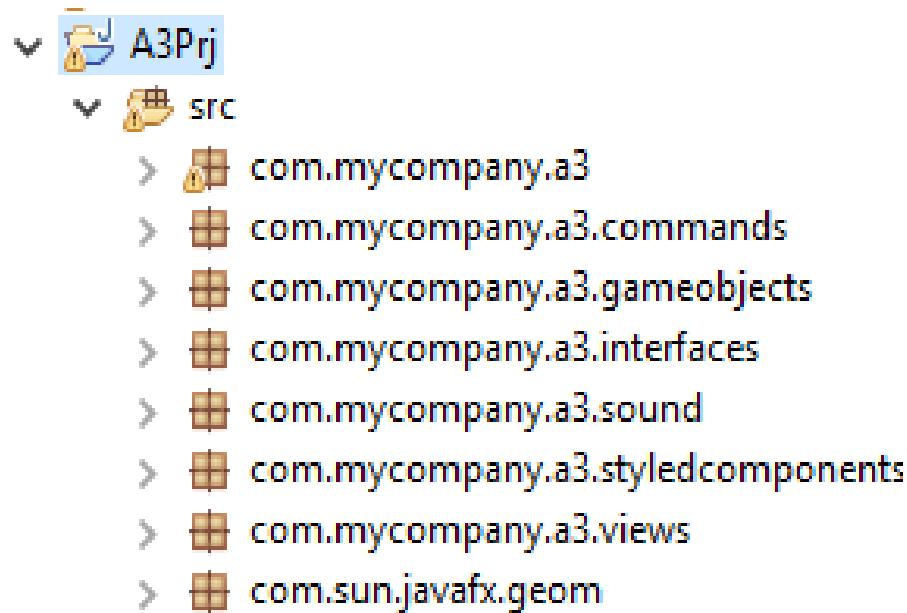
Java Packages

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



Java Packages (cont)

Example Only



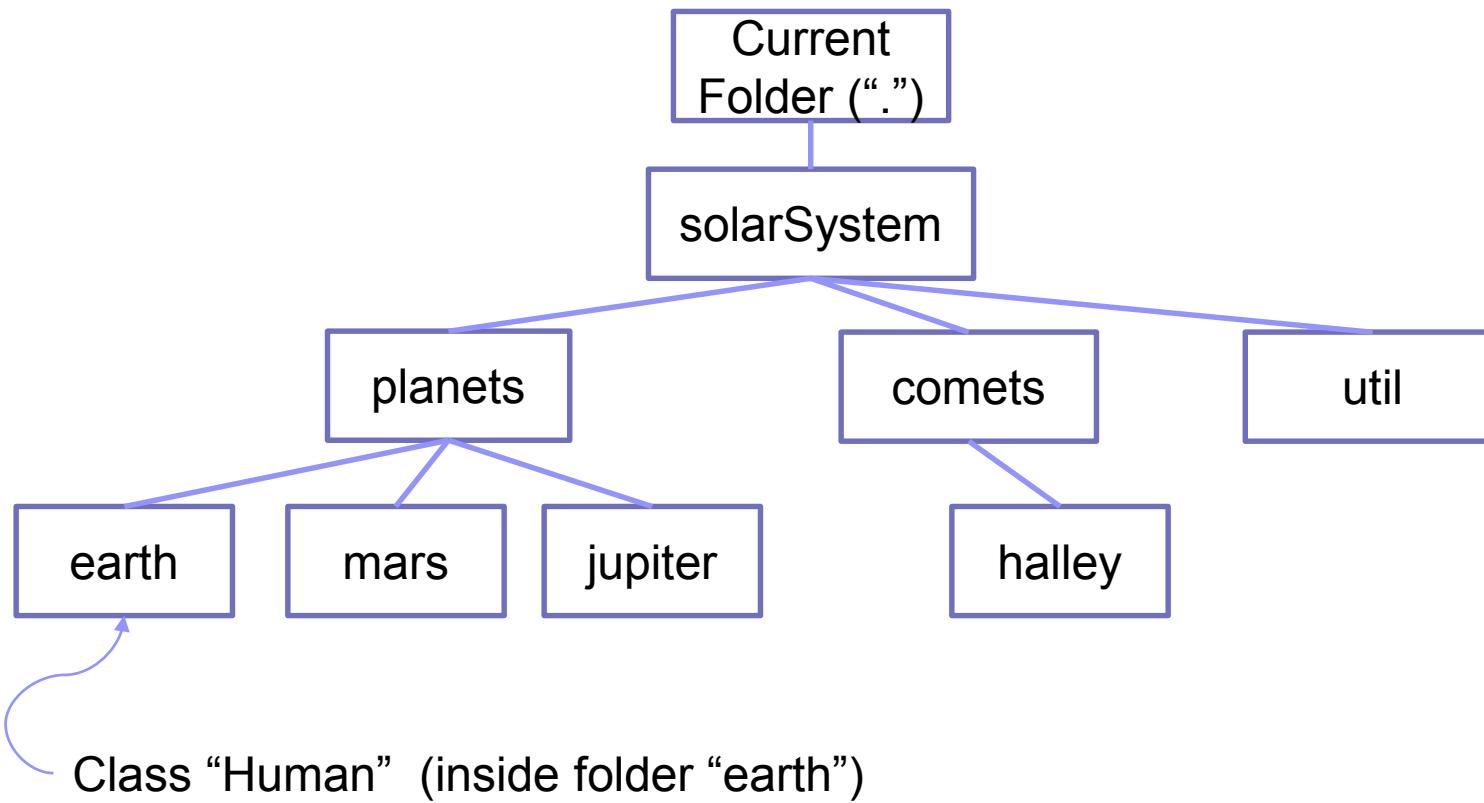
Java Packages (cont.)

- 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...  
}
```

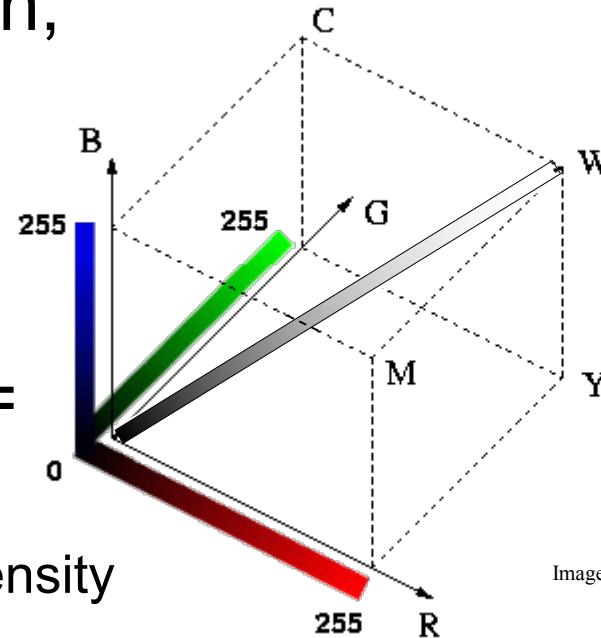
Packages and Folders

- 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



R: red
G: green
B: blue
C: cyan
M: magenta
Y: yellow
W: white

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]
```

- Questions:
- (1) Class name ?
 - (2) Methods invocation ?
 - (3) Capitalized GREEN ?

Breaking Encapsulations

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

```
public class Point {  
    public double x, y;      ← BAD!  
    public Point () {  
        x = 0.0 ;    y = 0.0 ;  
    }  
    // other methods here...  
}
```

Breaking Encapsulations (cont.)

- The correct way, with “Accessors”:

```
public class Point {  
    private double x, y ; ← Note  
  
    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.  
}
```

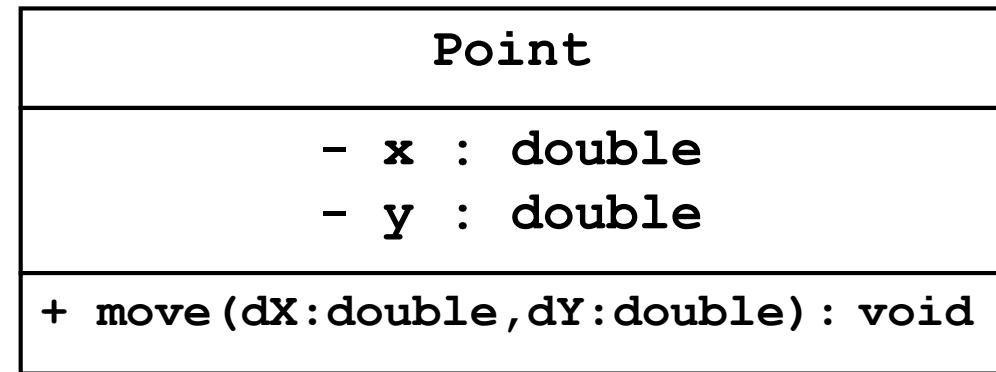
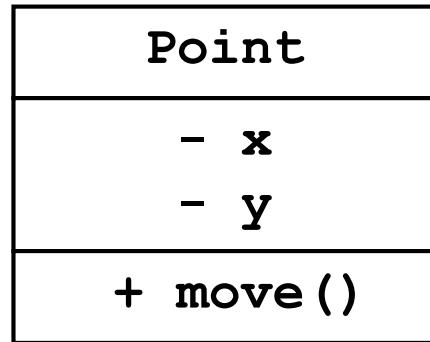
UML “Class Diagrams”

- A **class diagram** in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

Source: https://en.wikipedia.org/wiki/Class_diagram

UML “Class Diagrams” (cont)

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



Class Name, Attributes, Methods notation
+, -, #, ~ ?
20

Java Visibility UML Notation

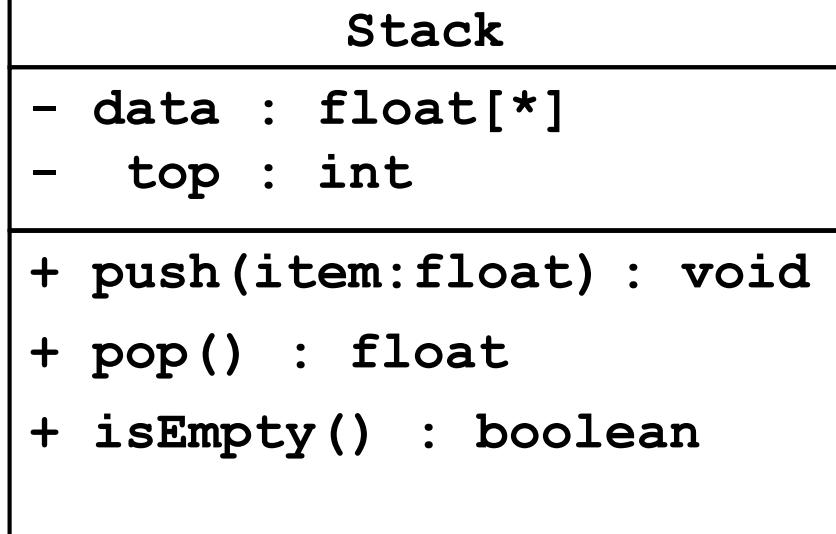
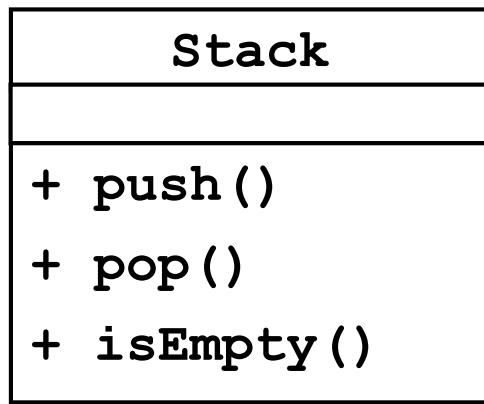
Java visibilityUML Notation

public	+
private	-
Protected	#
package	~

UML “Class Diagrams” (cont.)

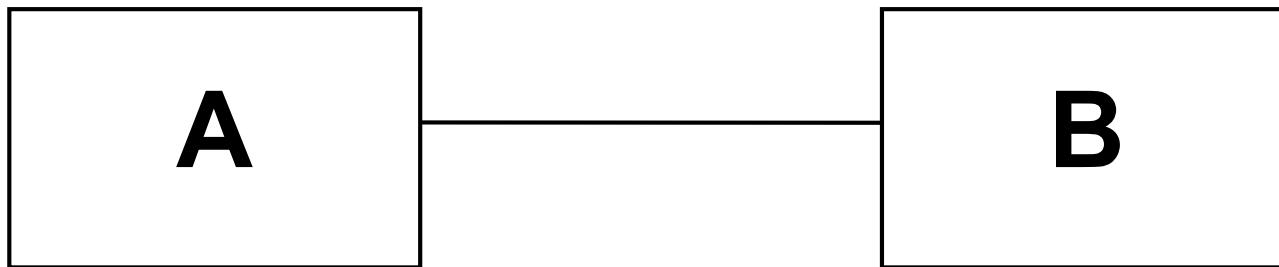
- o UML for the “**Stack**” class:

Stack



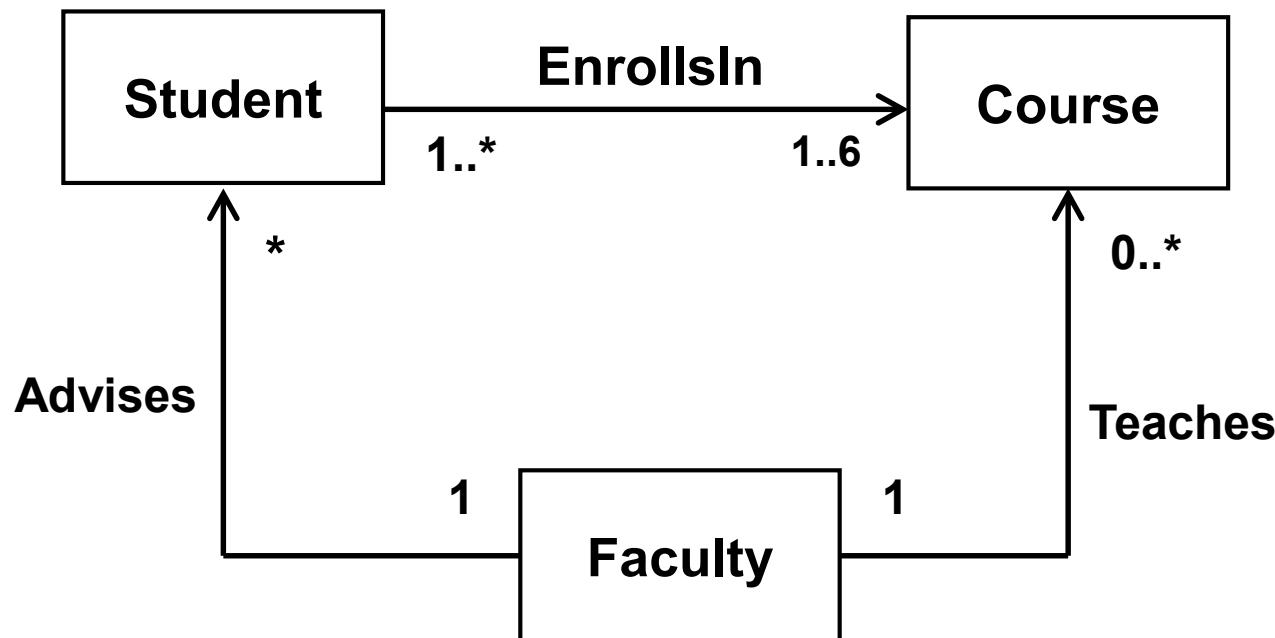
Associations

- 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)



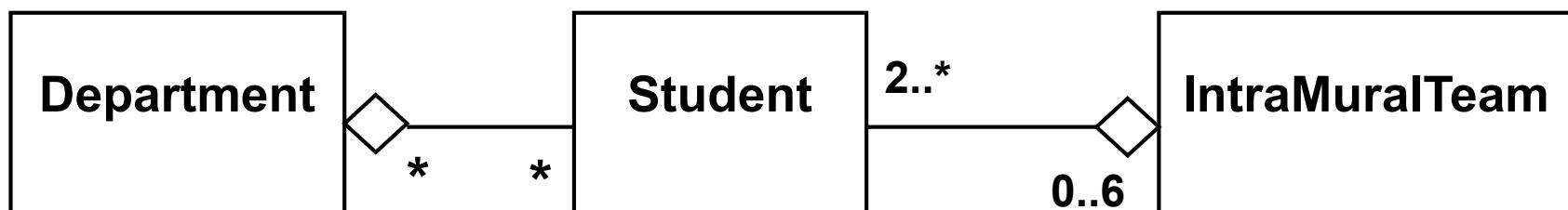
Multiplicity

0..1	No instances or one instance	A flight seat can have no or one passenger only
1	Exactly one instance	An order can have only one customer
0..* or *	Zero or more instances	A class can have zero or more students.
1..*	One or more instances (at least one)	A flight can have one or more passenger

Special Kinds Of Associations

- **Aggregation**

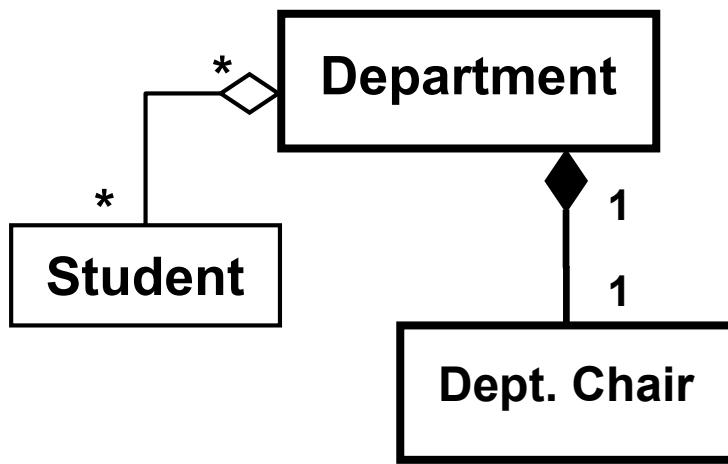
- Represents **"has-a"** or **"is-Part-Of"**



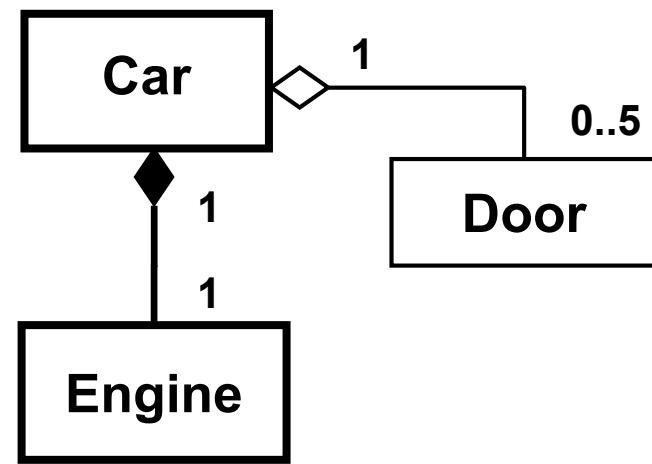
- An IntraMuralTeam is an aggregate of (has) 2 or more Students
- A Student *is-a-part-of* at most six Teams
- A Department has any number of Students
- A Student can belong to any number of Departments (e.g. double major)

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**)



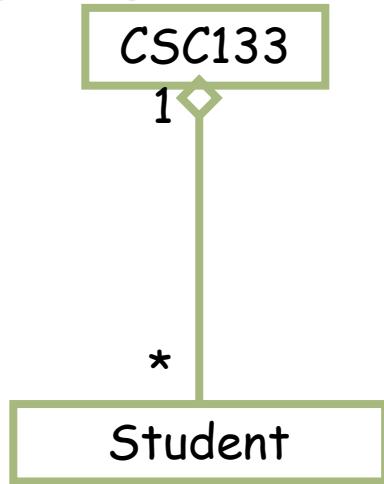
Exclusive ownership



Required ownership

Example: Aggregation vs. Composition

Aggregation



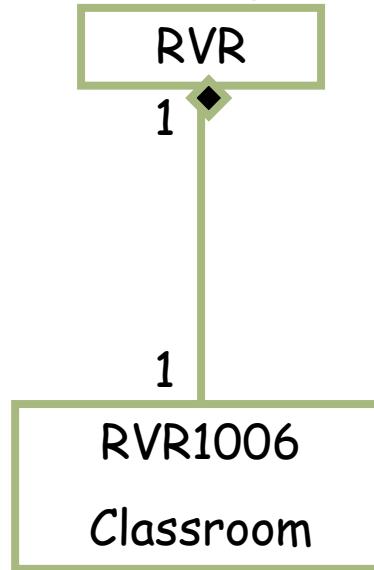
An association in which one class belongs to a collection

Shared: An object can exist in more than one collections

No ownership implied

Denoted by hollow diamond on the “contains” side

Composition



RVR =
Riverside Hall
ECS Building

An association in which one class belongs to a collection

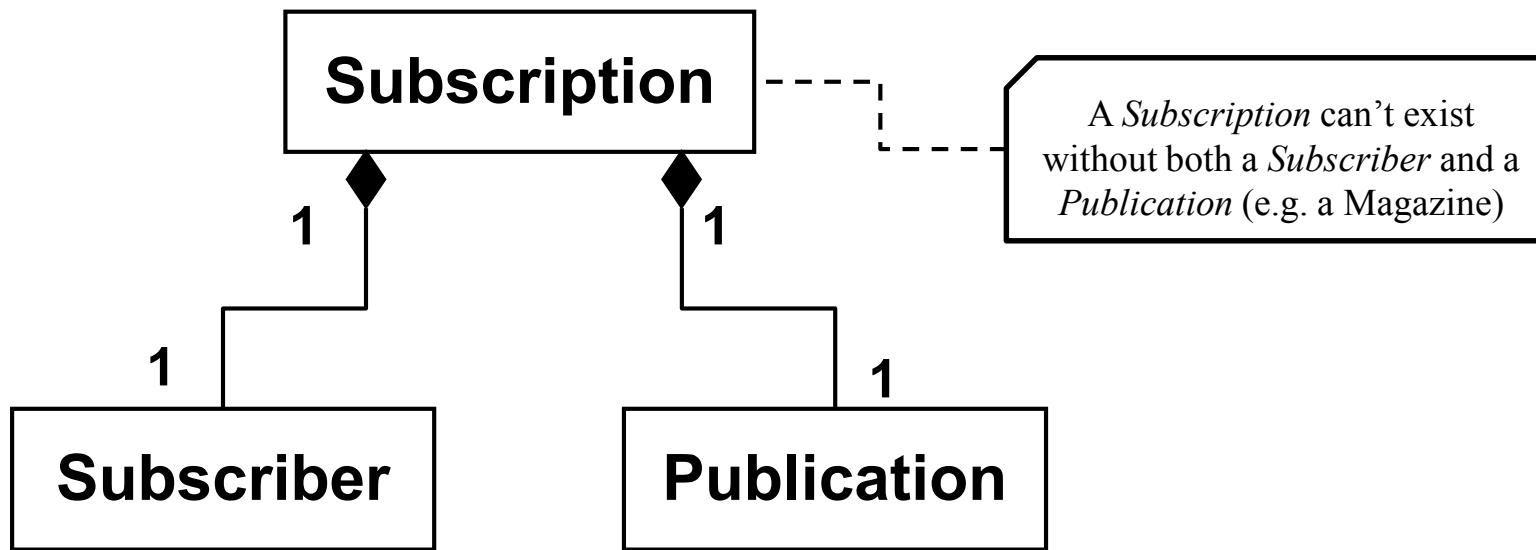
No Sharing: An object cannot exist in more than one collections

Strong “has a” relationship
Ownership

Denoted by filled diamond on the “contains” side

Special Kinds Of Associations (cont.)

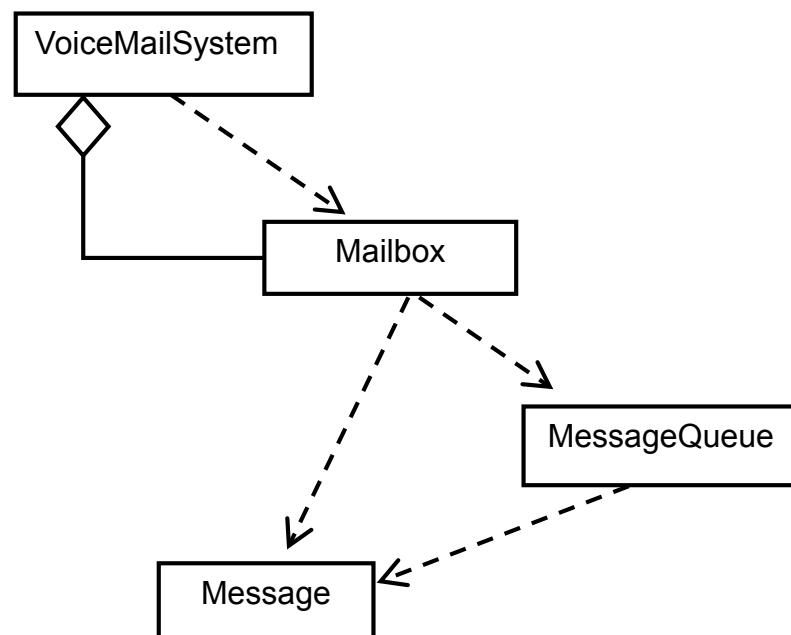
- Composition (another example)



Special Kinds Of Associations (cont.)

- Dependency
 - Represents “uses” (or “knows about”)

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



More on Dependency (cont.)

- **Dependency**

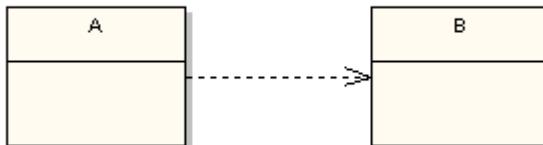


- Represents “uses” (or “knows about”)
- It means that the class at the source end of the relationship has some sort of dependency on the class at the target (arrowhead) end of the relationship.
- Class A uses class B, but that class A does not contain an instance of class B as part of its own state.

Examples Dependency (cont.)

```
class A {
    void foo(){
        b object= new B();
        object.baar();
    }
}

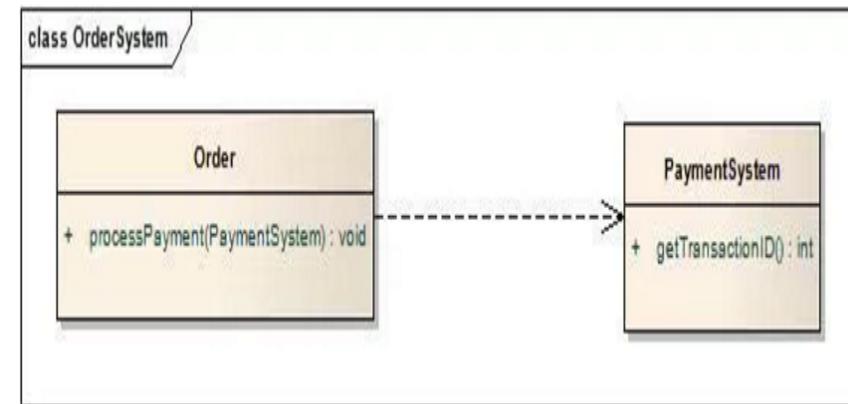
class B {
    void baar(){
    }
}
```



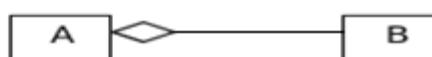
Class A uses class B. Therefore class A has a dependency on class B.

```
public class PaymentSystem {
}

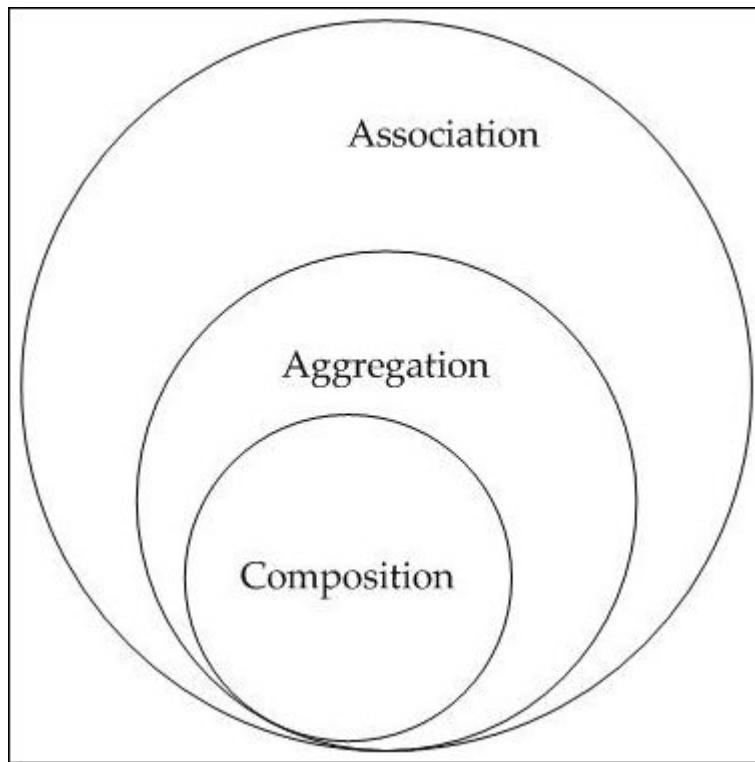
public class Order {
    public void processPayment(PaymentSystem ps){
    }
}
```



Recap 1

Relationship	Depiction	Interpretation
Dependency		<p>A depends on B. In Java we can consider the dependency relationship if the source class has a reference to the dependent class directly or source class has methods through which the dependent objects are passed as a parameter or refers to the static operation's of the dependent class or source class has a local variable referring to the dependent class etc.</p>
Association		<p>An A sends messages to a B. Associations imply a direct communication path. In programming terms, it means instances of A can call methods of instances of B, for example, if a B is passed to a method of an A.</p>
Aggregation		<p>An A is made up of B. This is a part-to-whole relationship, where A is the whole and B is the part. In code, this essentially implies A has fields of type B.</p>
Composition		<p>An A is made up of B with lifetime dependency. That is, A aggregates B, and if the A is destroyed, its B are destroyed as well.</p>

Recap 2



	Aggregation	Composition
Life time	Have their own lifetime	Owner's life time
Relation	Has	part-of
Example	Car has driver	Engine is part of Car

Sometimes, it can be a complicated process to decide if we should use association, aggregation, or composition. This difficulty is caused in part because **aggregation** and **composition** are subsets of **association**, meaning they are specific cases of association.

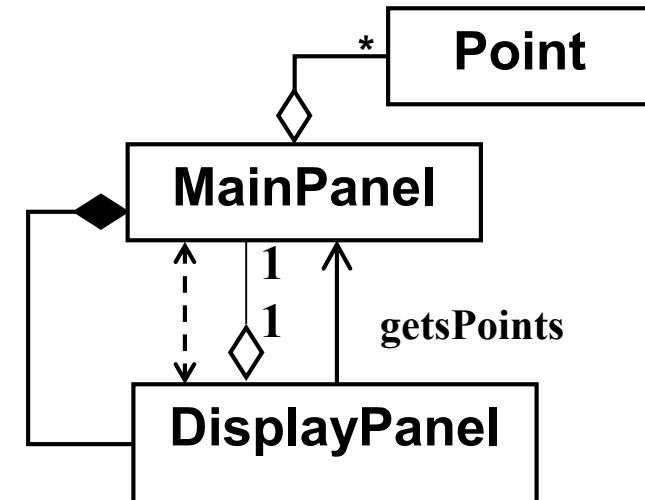
Source: <https://softwareengineering.stackexchange.com/questions/61376/aggregation-vs-composition>

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 ;
    }
    ...
}
```



Question: Code reference to diagram ?

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 {

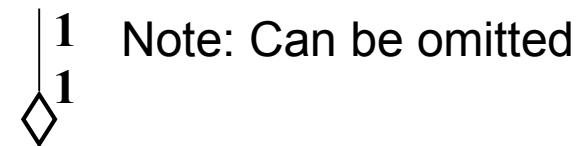
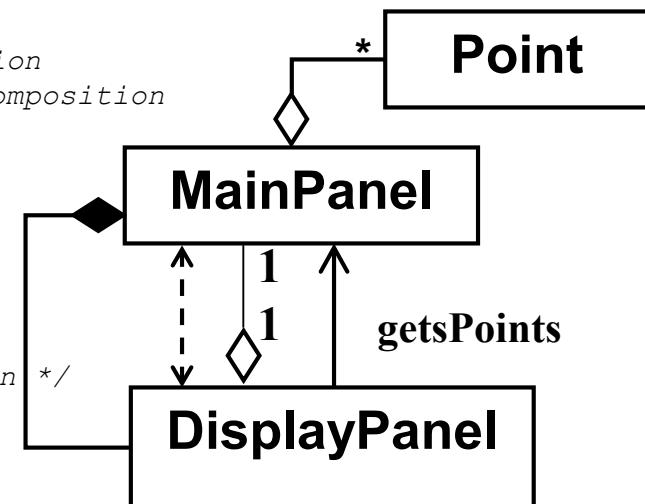
    private ArrayList<Point> myPoints ;           //my Point aggregation
    private DisplayPanel myDisplayPanel;           //my DisplayPanel composition

    /** 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 {

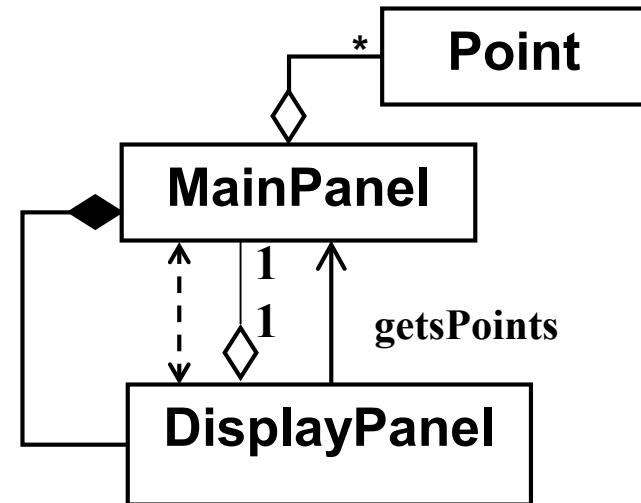
    private MainPanel myMainPanel;

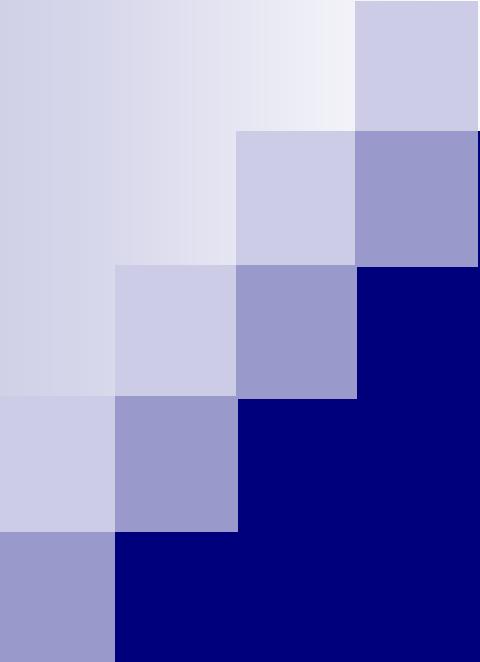
    public DisplayPanel(MainPanel m) {

        //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

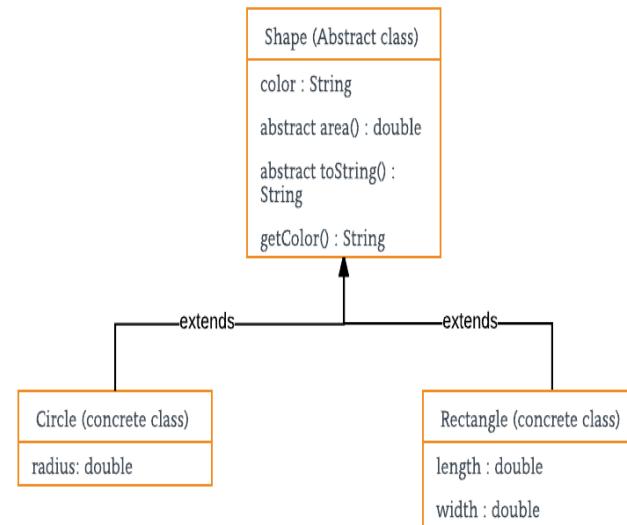
Computer Science Department
California State University, Sacramento

Overview

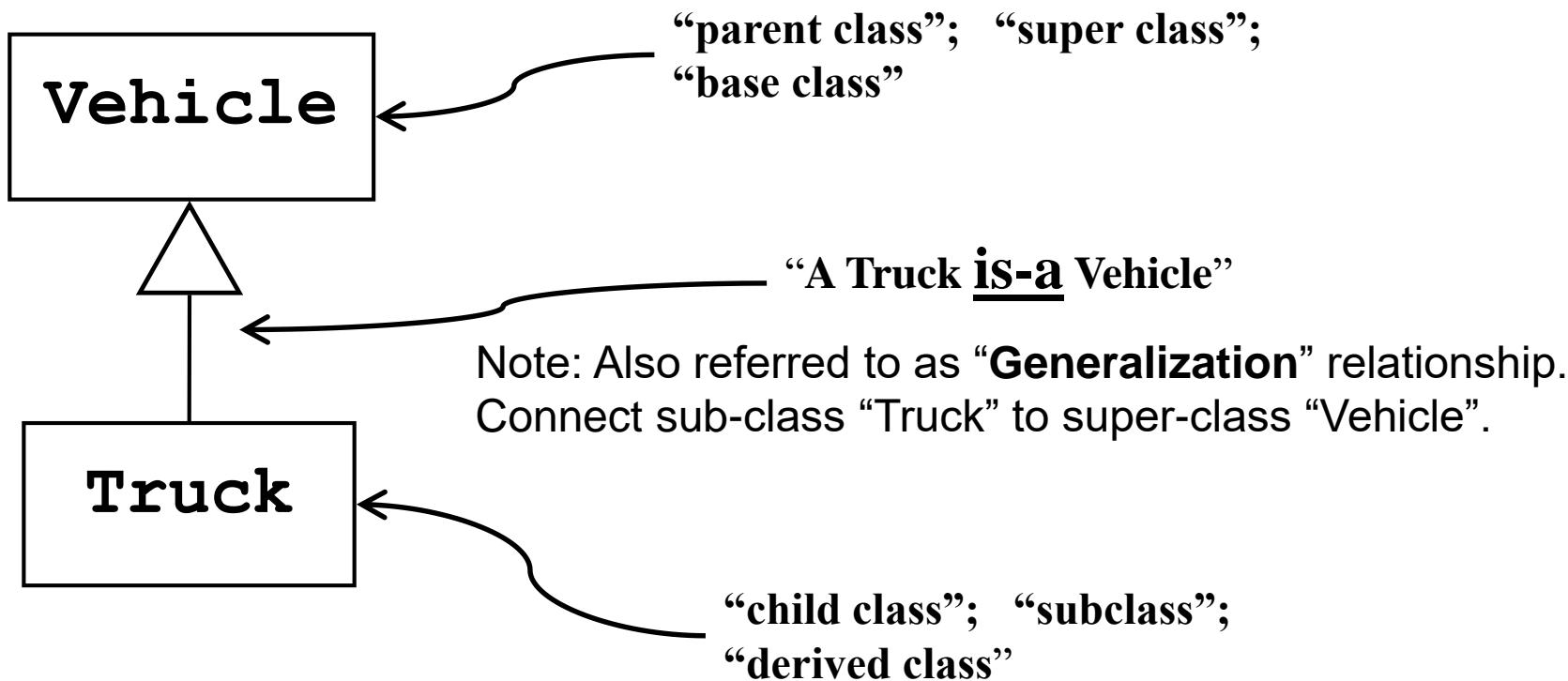
- **Definition**
- **Representation in UML, Implementation in Java,
The “IS-A” concept**
- **Inheritance Hierarchies**
- **Overriding, Overloading**
- **Forms of Inheritance: Extension, Specialization,
Specification**
- **Implications for Public vs. Private data**
- **Abstract classes and methods**
-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

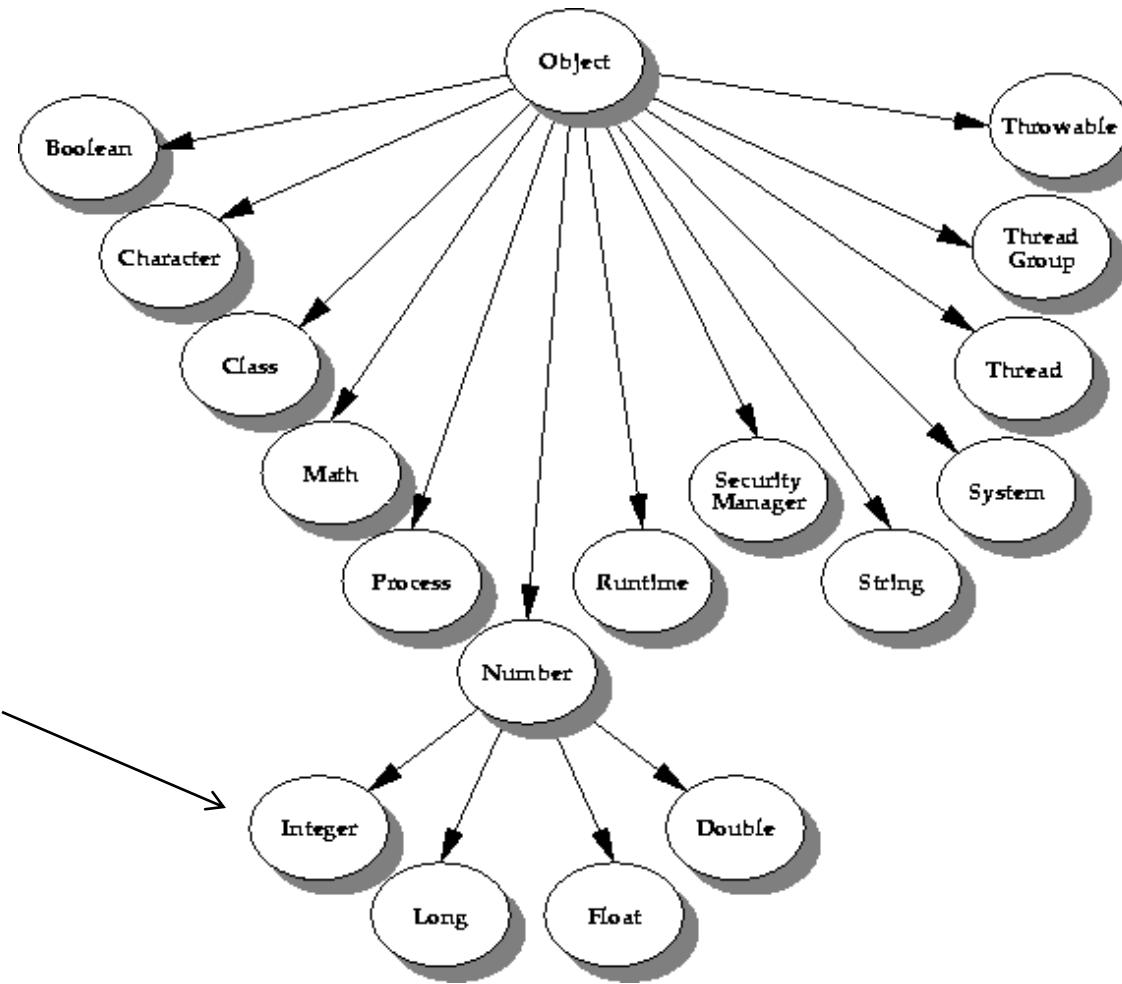
- Specified with the keyword “extends” :

```
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”

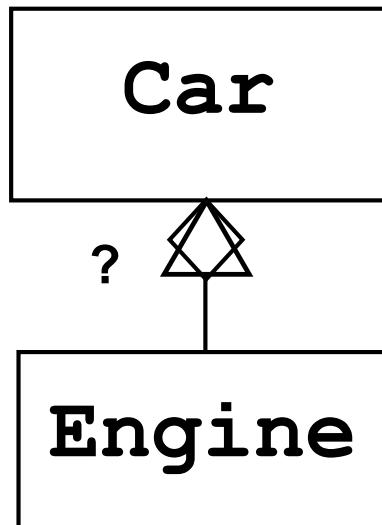
Recall Object: The Cosmic Superclass



The `java.lang` package contains the collection of base types (language types) that are always imported into any given compilation unit. This is where you'll find the declarations of `Object` (the root of the class hierarchy) and `Class`, plus threads, exceptions, wrappers for the primitive data types, and a variety of other fundamental classes.

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

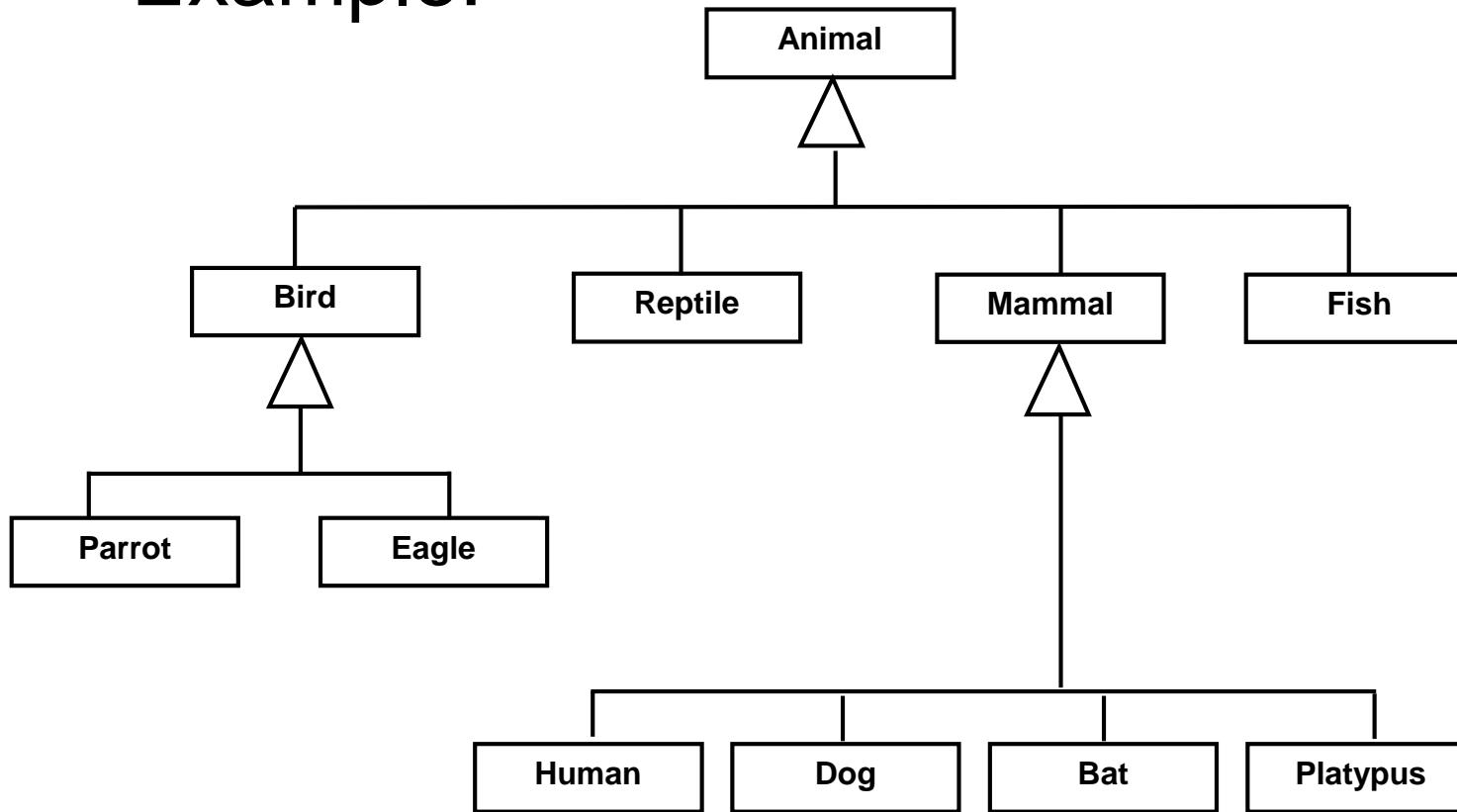
A Car “is an” Engine ? X

A Car “has-an” Engine ✓

An Engine “is a part of” a Car ✓

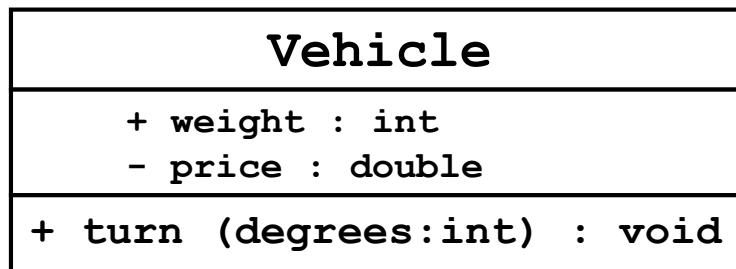
Inheritance Hierarchies

- Example:



Method Overriding

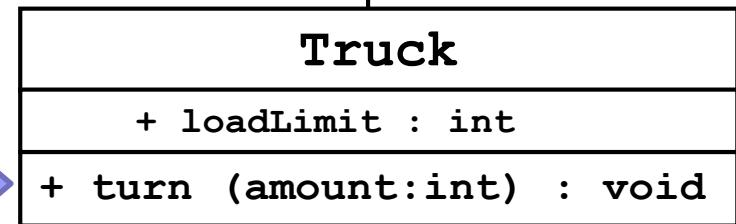
- Inheritance leads to an interesting possibility:
duplicate method declarations



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

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

    ...
}
```



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

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

    ...
}
```

Truck's turn(int) “***overrides***”

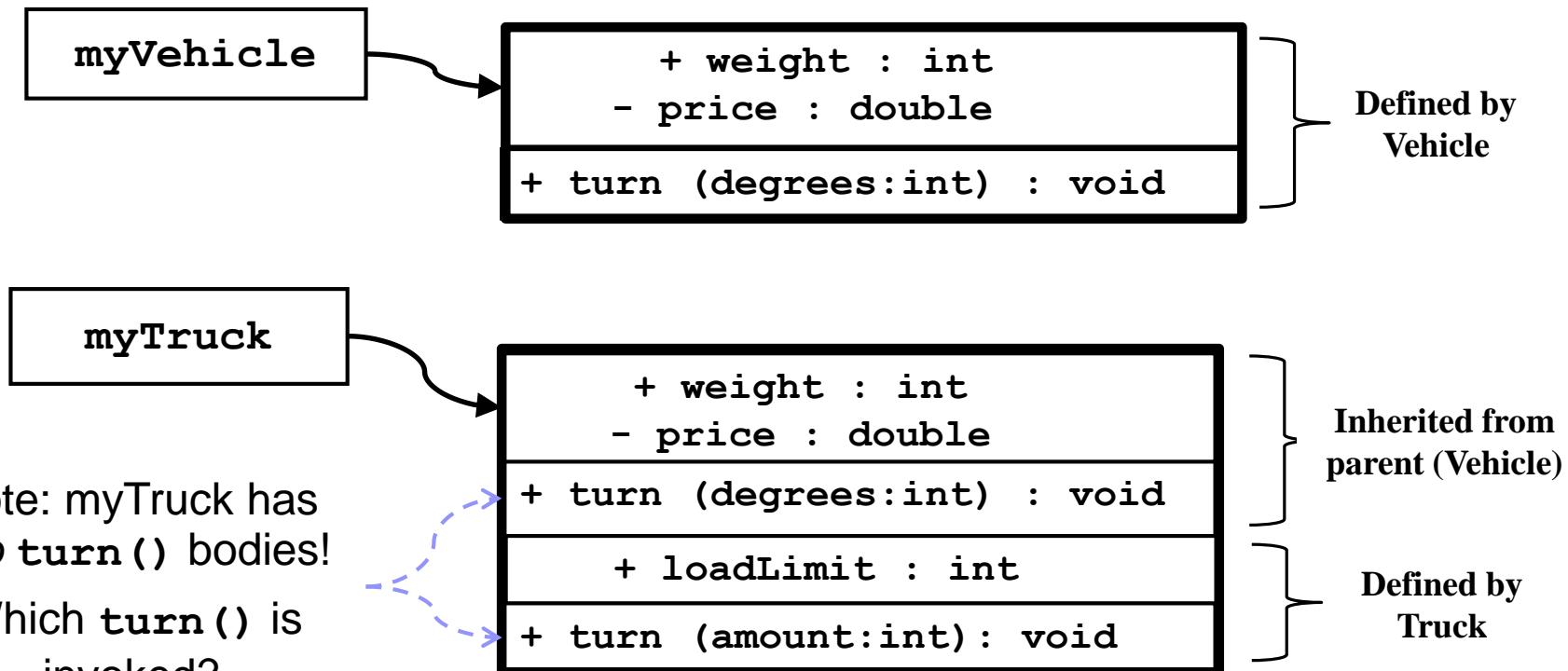
Vehicle's turn(int)

Effects of Method Overriding

Consider the following code:

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

... then we get two objects:



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

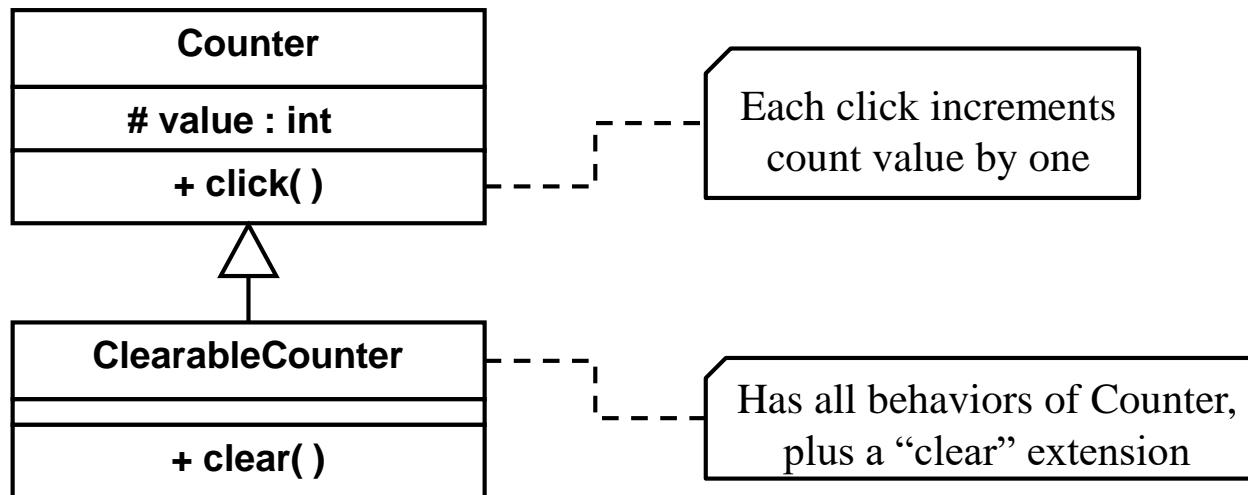
- **Not the same as “overriding”...**
 - **Overloading == same name but different parameter types**
 - **Can occur *in the same class or split between parent/child classes***
- **Overloading examples:**
 - Methods with different numbers of parameters:
`distance(p1); distance(p1, p2)`
 - Constructors 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);`

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)

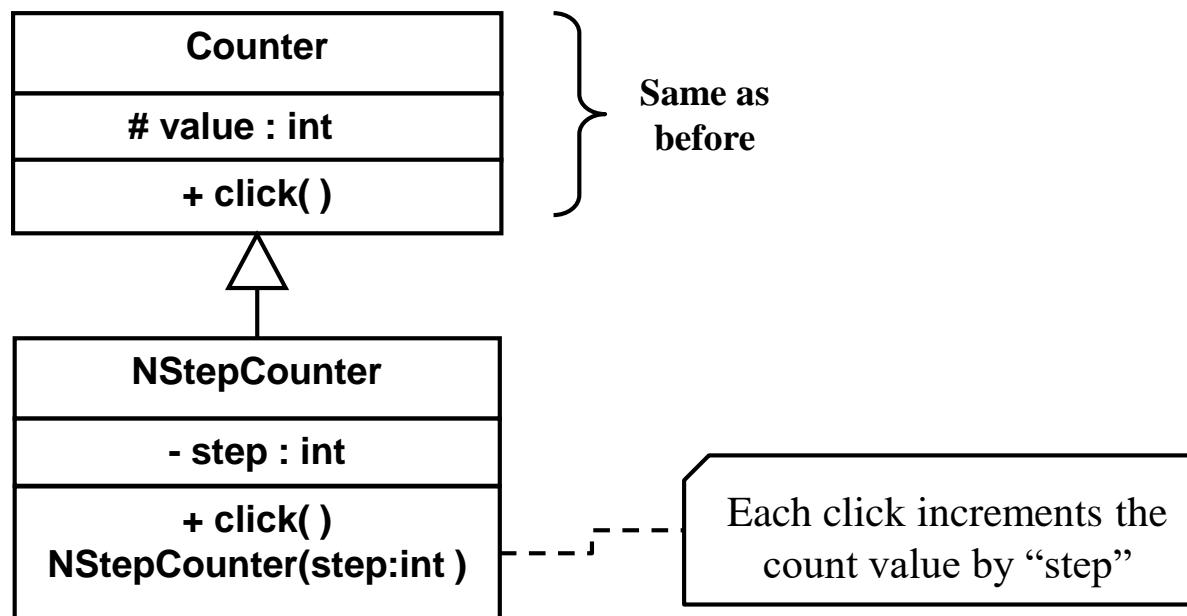


Inheritance for Extension (cont.)

```
/** 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

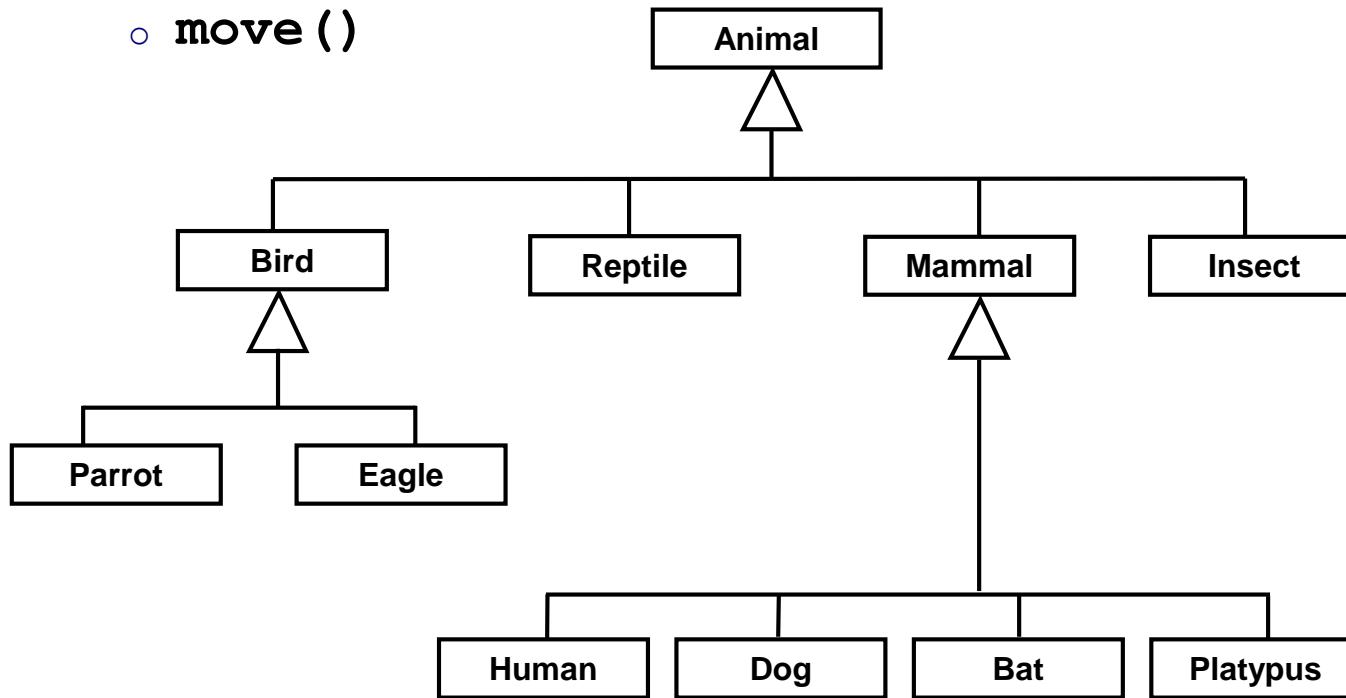


Inheritance for Specification

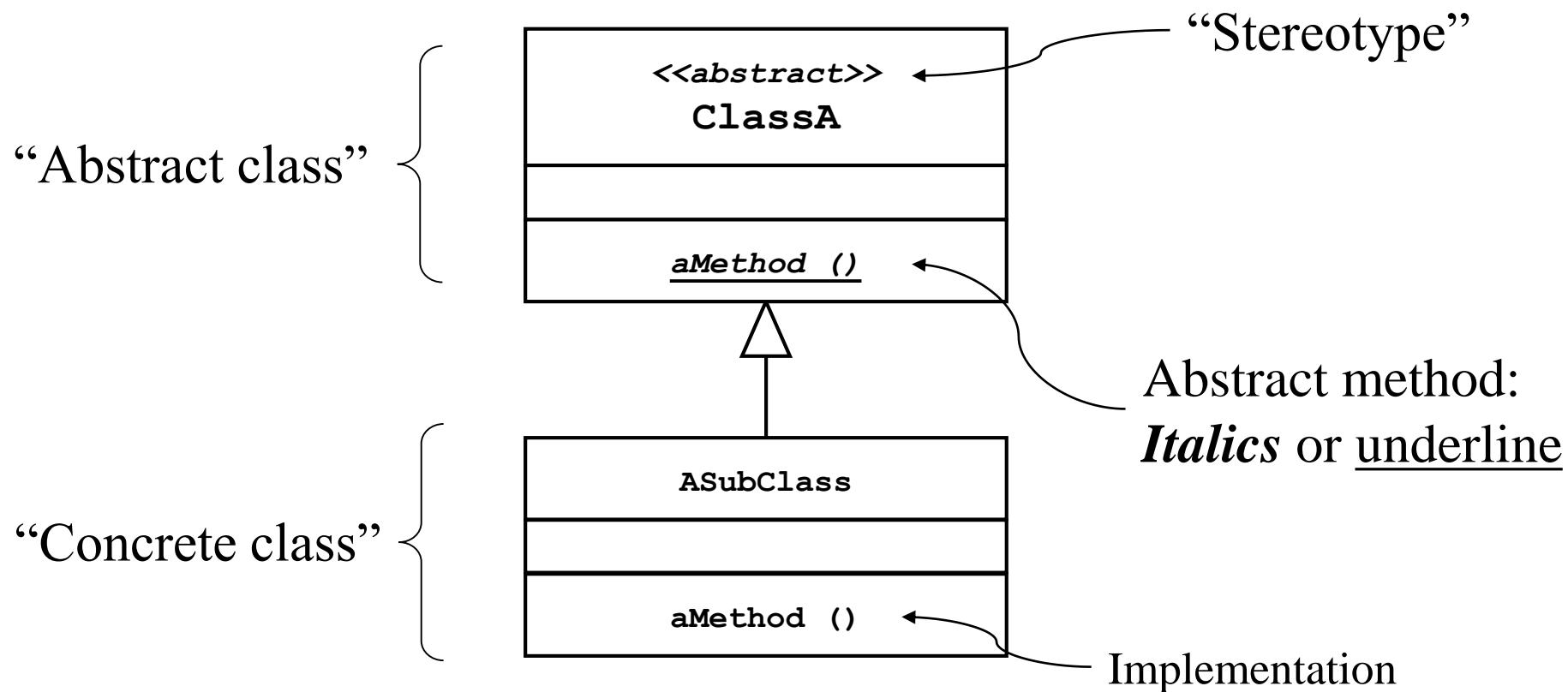
- Used to *specify (define)* behavior declared (but not defined) by the parent
 - Classes which declare but don't define behavior:
Abstract Classes
 - Methods which don't contain implementations:
Abstract methods

Abstract Classes & Methods

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

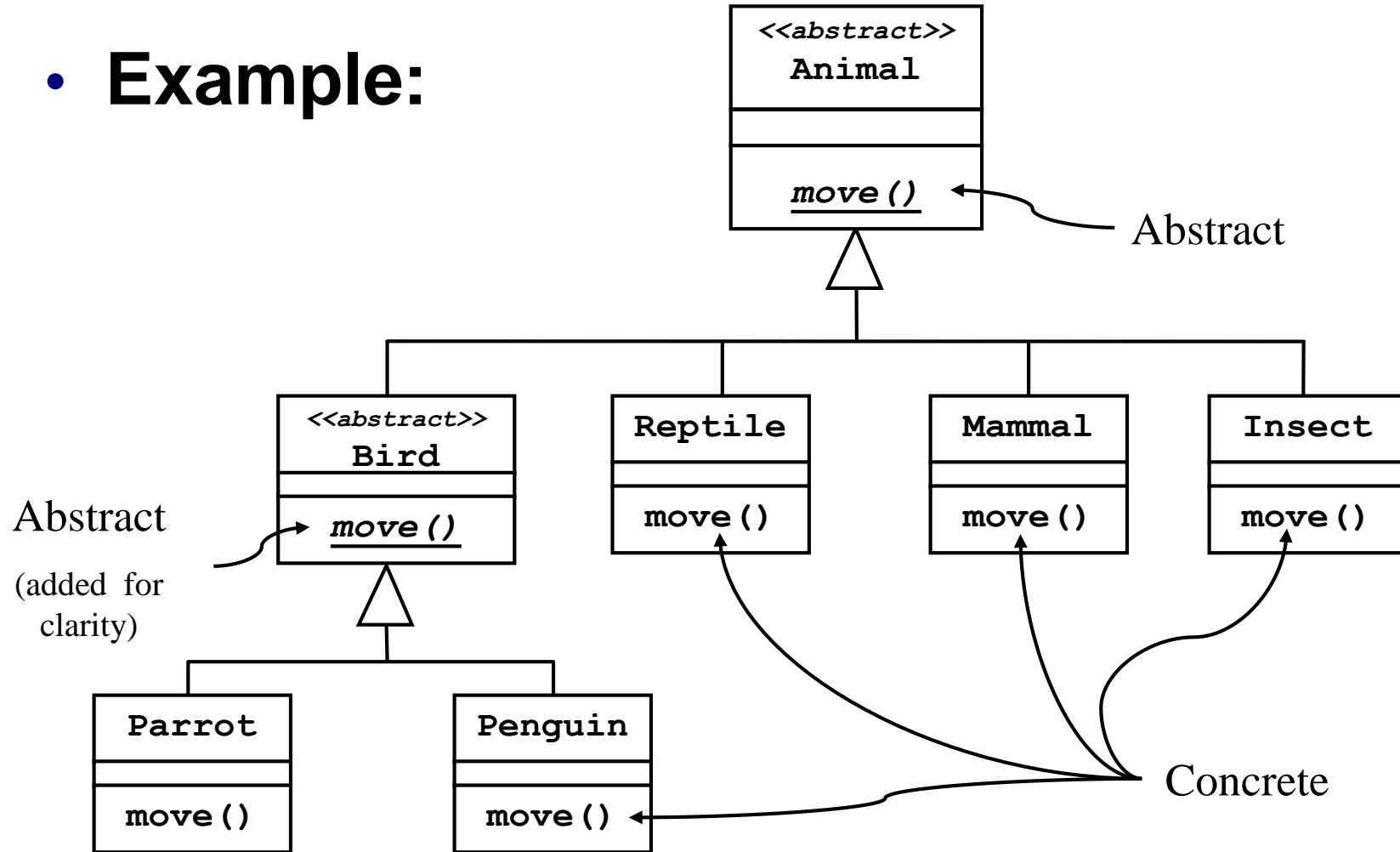


Inheritance for Specification (cont.)



Inheritance for Specification (cont.)

- Example:



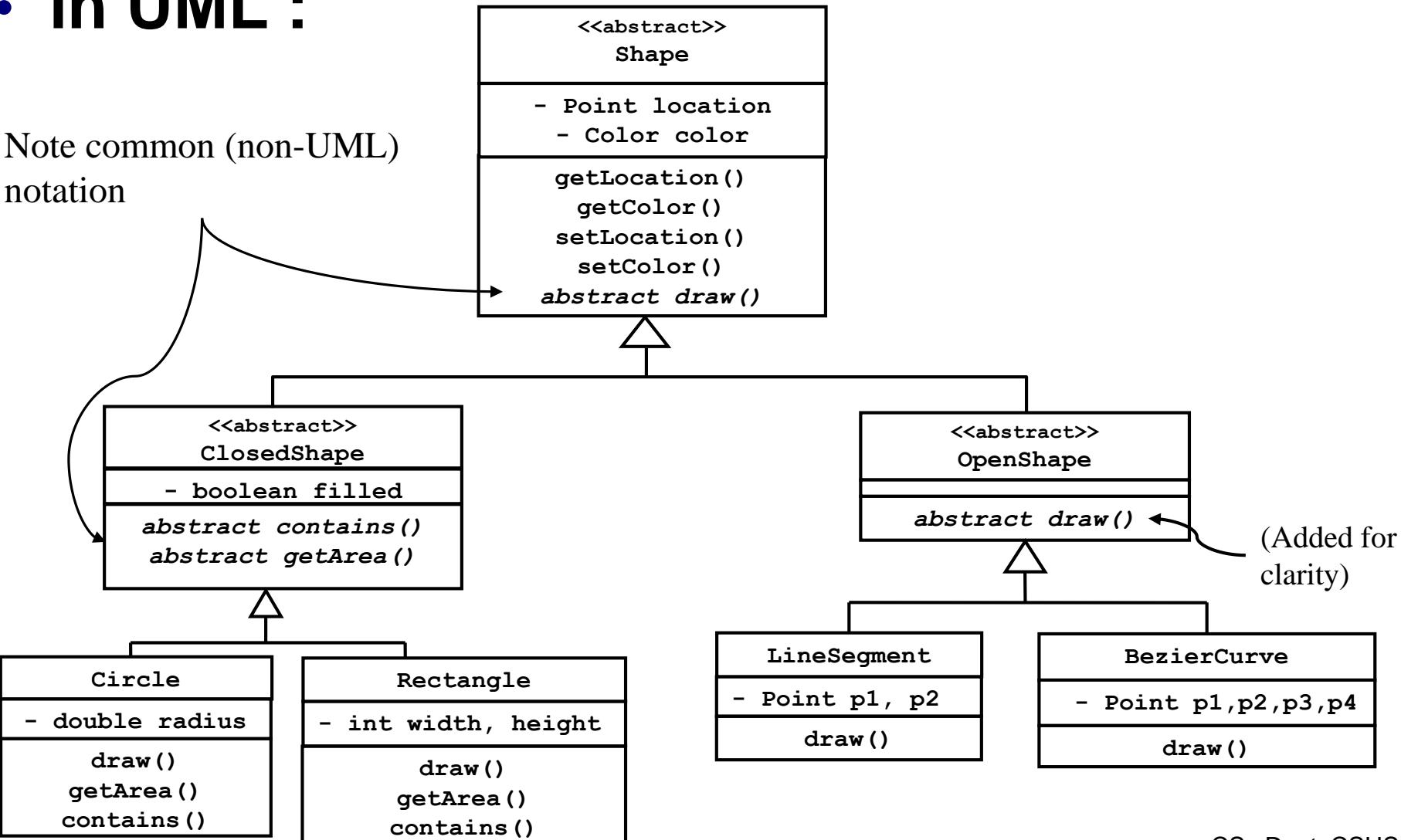
Inheritance for Specification (cont.)

- 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 :

Note common (non-UML) notation



Implications for Public vs. Private data

recall, from the encapsulation section:

Point (without “Accessors”):

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

BAD

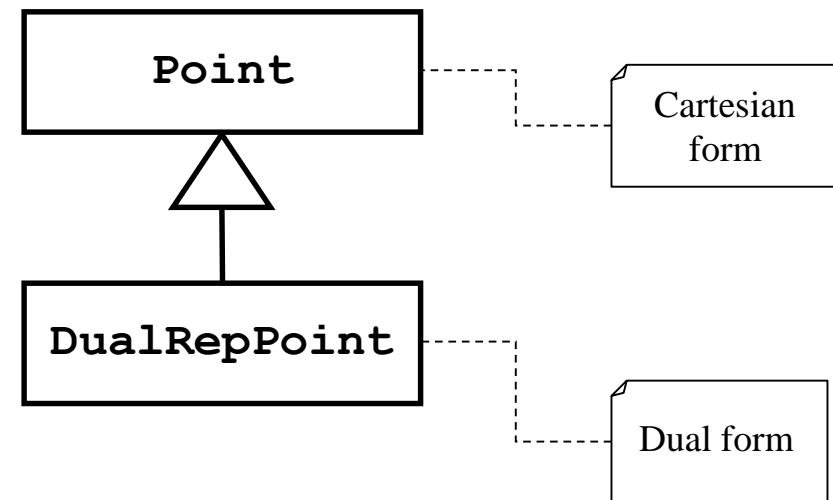
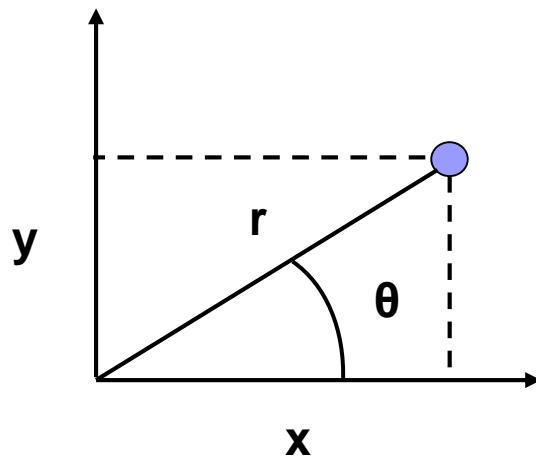
Point (with “Accessors”):

```
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

Now we will learn why!

Example: extend “Point” to create “DualRepPoint”



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
```

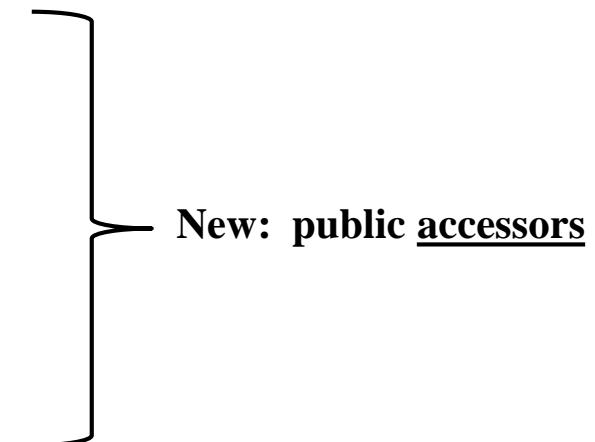
```
    private void updateRectangularValues() {
```

```
        x = radius * Math.cos(Math.toRadians(angle)) ;
```

```
        y = radius * Math.sin(Math.toRadians(angle)) ;
```

```
}
```

```
}
```



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!
(Calling parent
setX/setY)**
- **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 ...

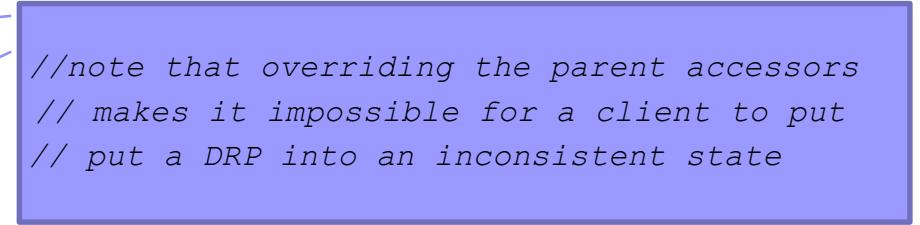
    // Override inherited accessors
    public void setX (double xVal) {
        super.setX(xVal) ;
        updatePolarValues() ;
    }

    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) ;
    }
}

```



//note that overriding the parent accessors
// makes it impossible for a client to put
// put a DRP into an inconsistent state

Java Abstract Classes

- Both classes and methods can be declared abstract

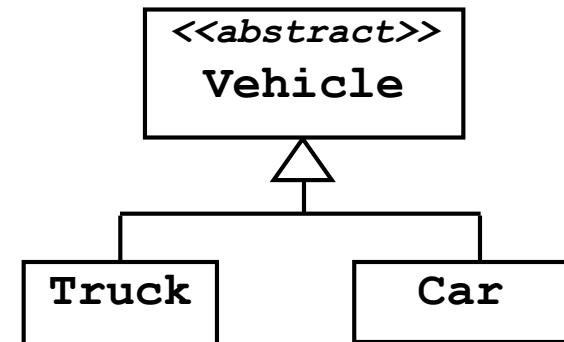
```
public abstract class Animal {  
    public abstract void move () ;  
}
```

- 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);
    }

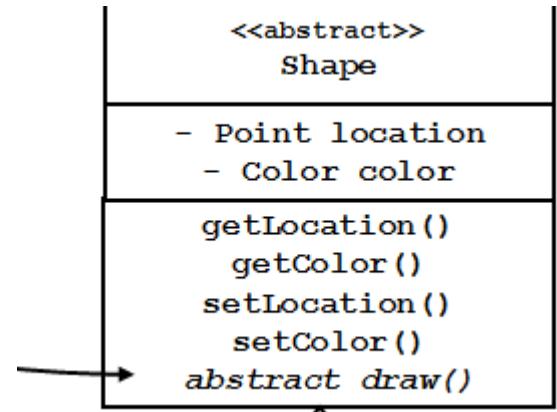
    public Point getLocation() {
        return location;
    }

    public int getColor() {
        return color;
    }

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

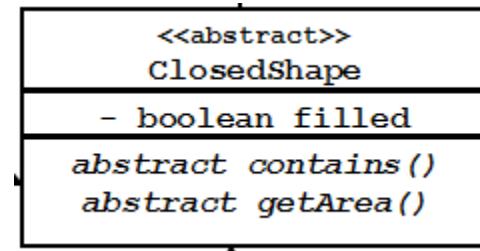
    public void setColor (int newColor) {
        color = newColor;
    }

    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 {

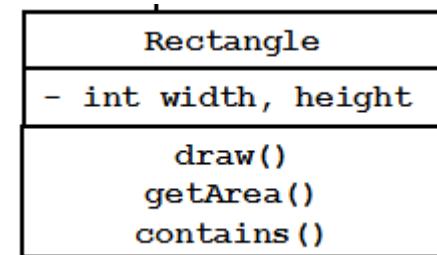
    private int width;
    private int height;

    public Rectangle() {
        super(true); //no-arg constructor of its parent (ClosedShape) is not called
        width = 2;
        height = 1;
    }

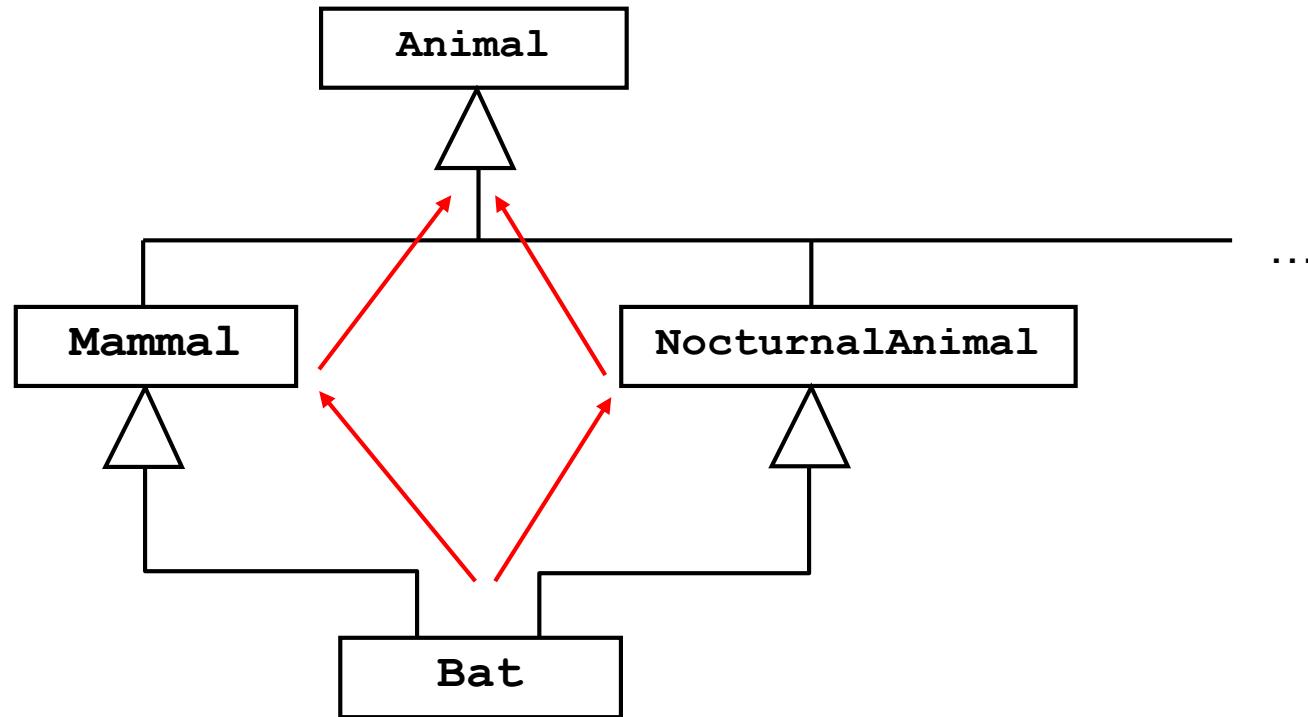
    public boolean contains(Point p) {
        //... code here to return true if p lies inside this rectangle,
        // or return false if not.
    }

    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(); // Specify a hierarchical path
}
```

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
 - H₂O: 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 ;
```

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

“Static” Polymorphism (cont.)

Another example: Method overloading:

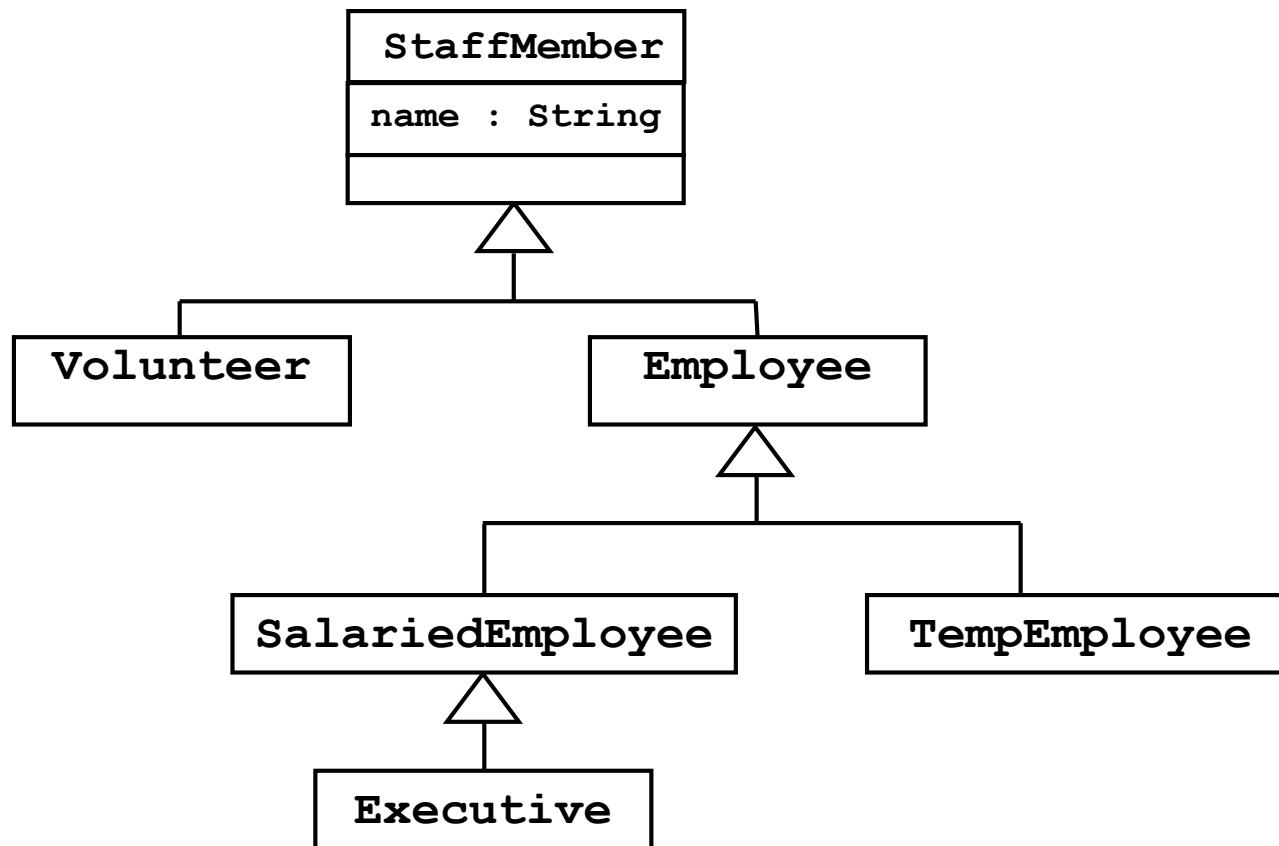
```
//return the distance to an origin
double distance (int x, int y) { . . . }

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

- o Same method name, for two different operations
- o “**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];
```

```
    . . .
```

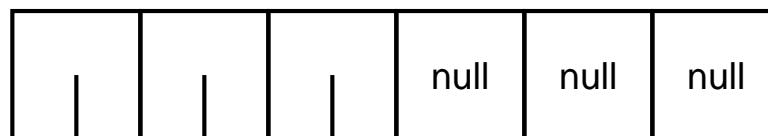
```
    staffList[0] = new SalariedEmployee ("Sam");
```

```
    staffList[1] = new Executive ("John");
```

```
    staffList[2] = new Volunteer ("Doug");
```

```
    . . .
```

staffList



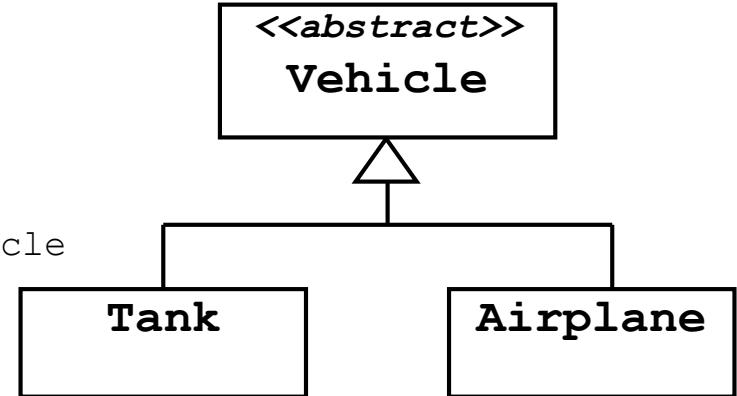
Sam John Doug

SalariedEmpl Executive Volunteer

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
```

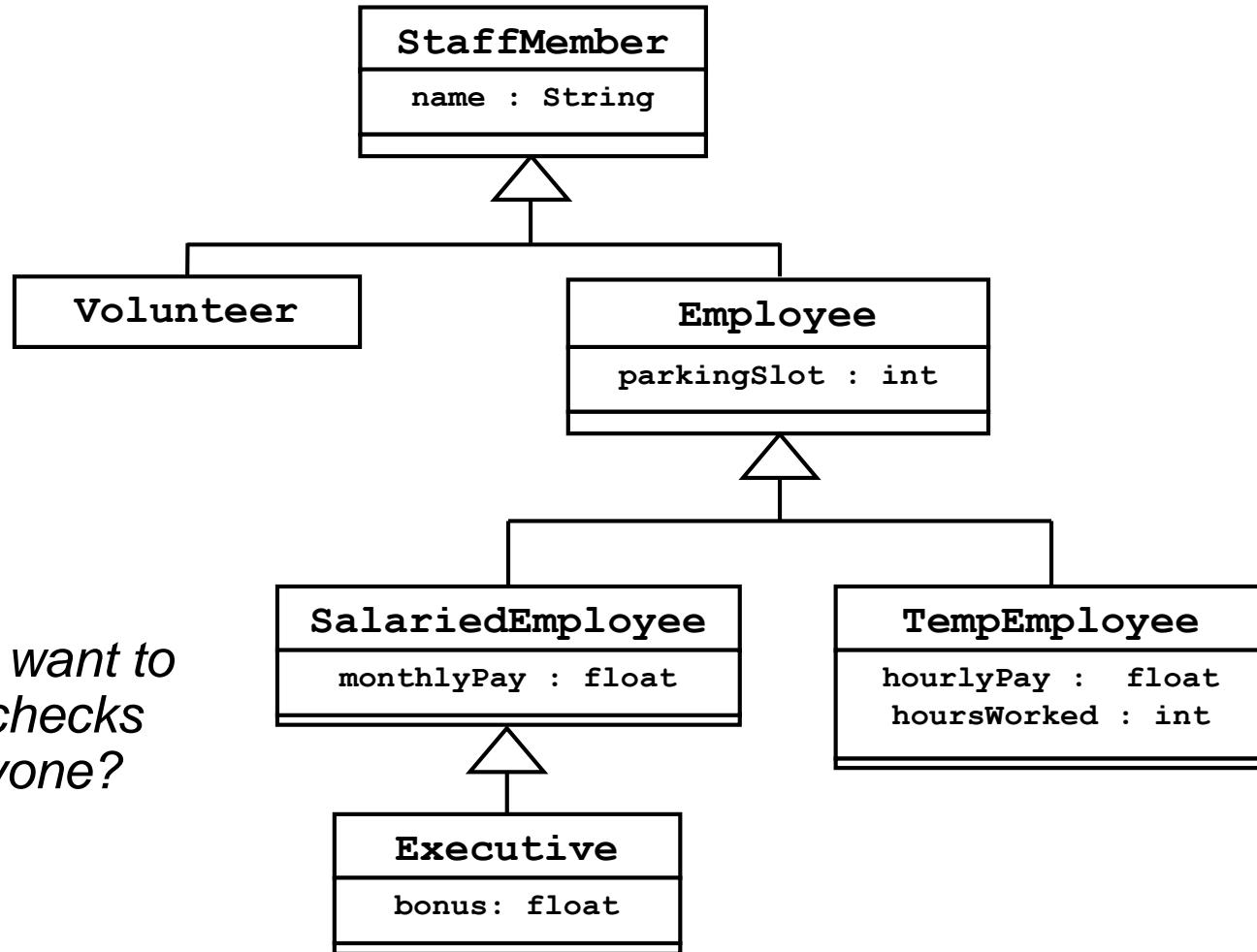


- “Downcasting” requires casting:

```
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?

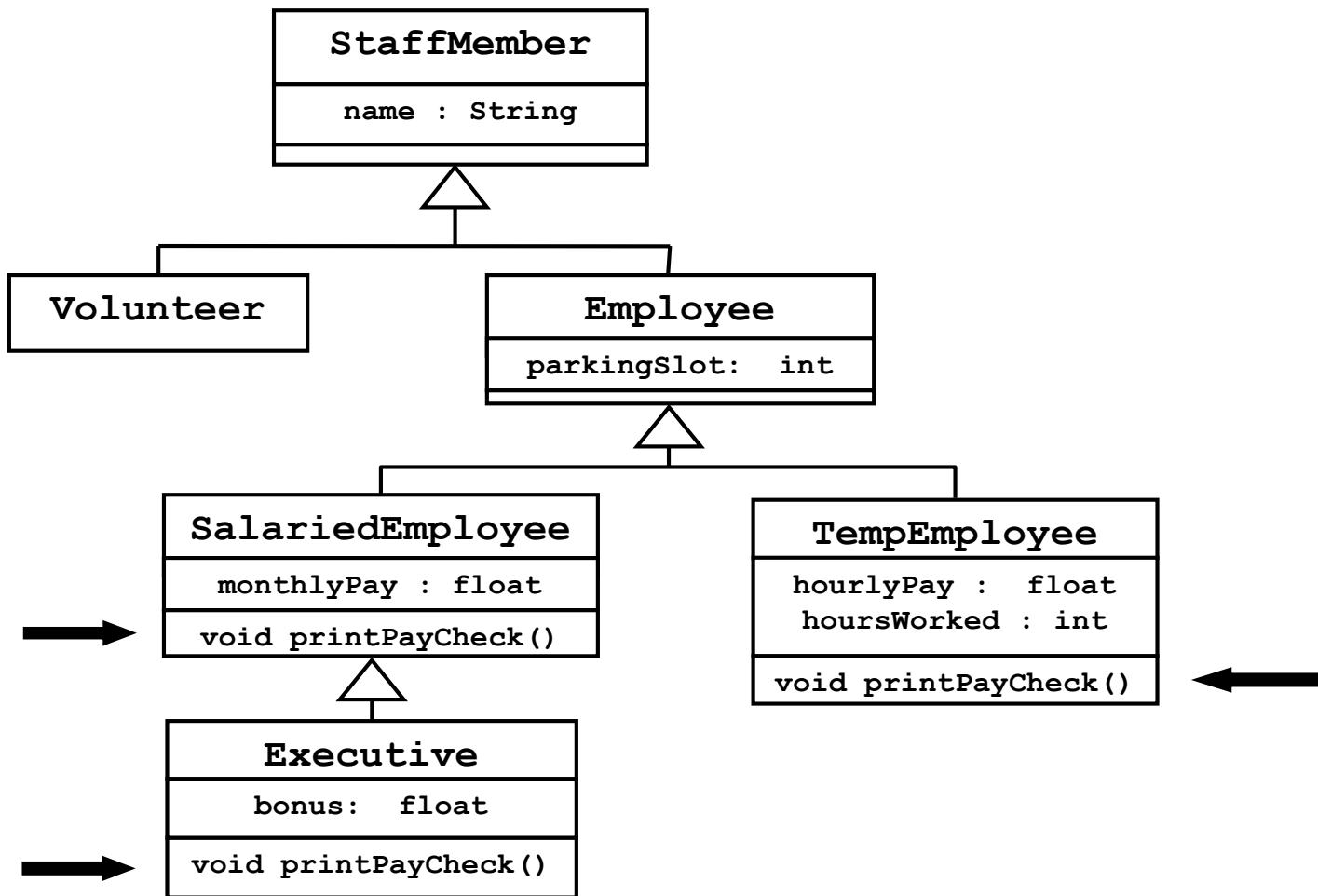
Runtime Polymorphism (cont.)

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() ;
}
...
```

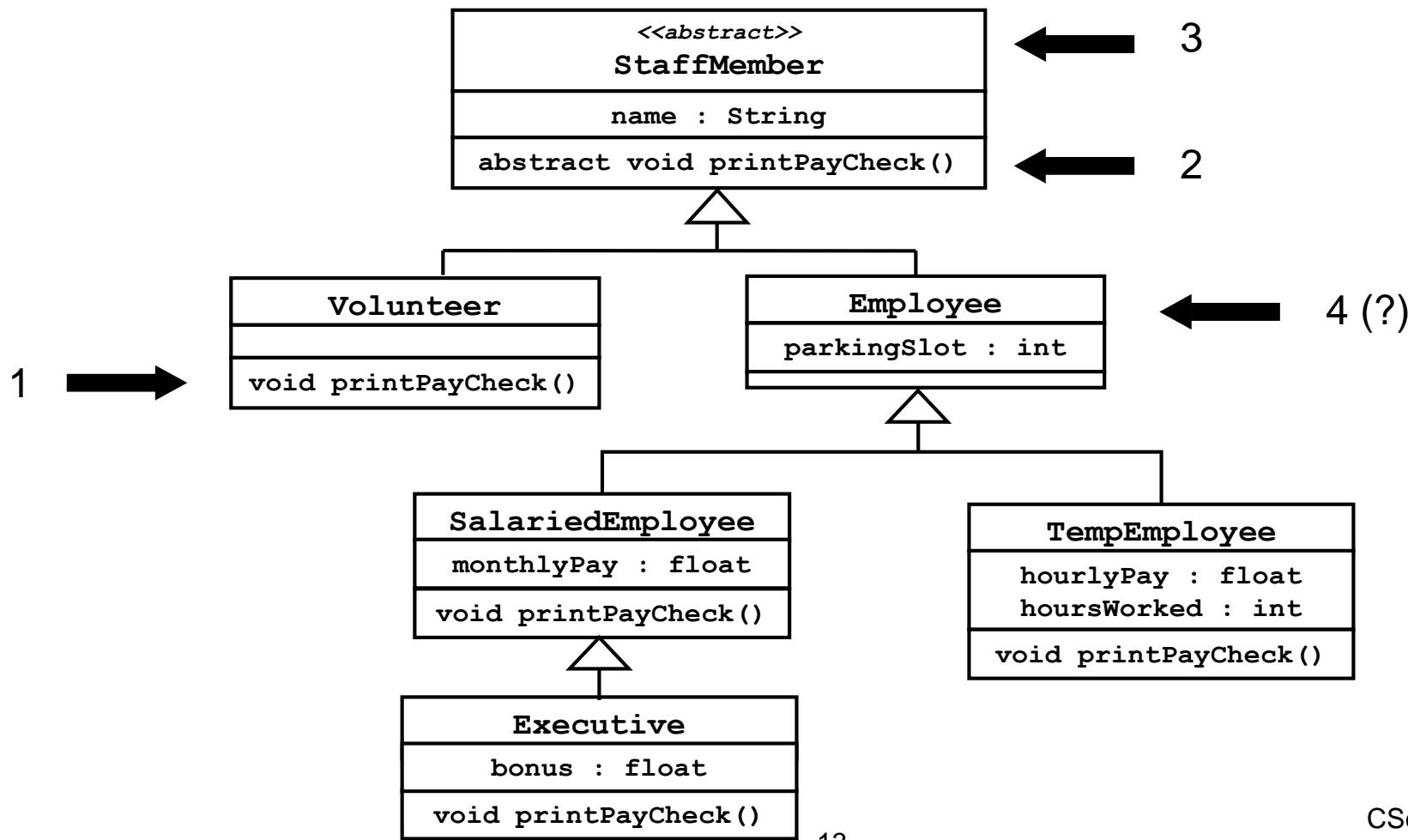
Now, the Print method which gets invoked is:

- *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

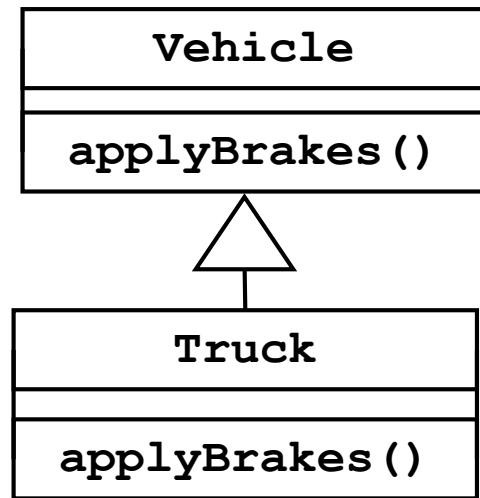
Ideally, every class should know how to deal with “printPayCheck” messages:



Polymorphism: Java

- **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**”

Java : Example



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 : Example (cont.)

Java

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

Output

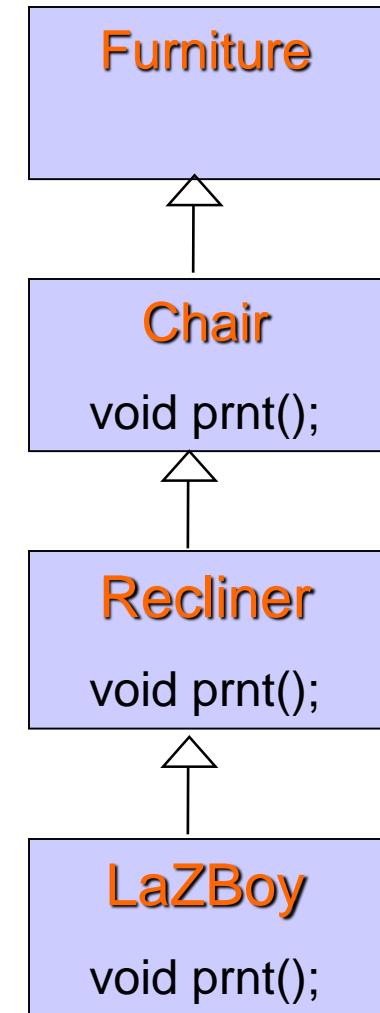
```
Applying truck brakes...  
Applying truck brakes...
```

Recap: Basis of Polymorphism

1. Inheritance
2. Method overriding
3. Polymorphic assignment // `SuperClassVariable = SubclassObject;`
4. Polymorphic methods
 - In Java, all methods are polymorphic.
 - That is, choice of method depends on the object.

Quiz: Polymorphism

```
abstract class Furniture {  
    public int numlegs;  
}  
  
abstract class Chair extends Furniture {  
    public String fabric;  
    abstract void prnt();  
}  
  
class Recliner extends Chair {  
    void prnt() {  
        System.out.println("I'm a recliner");  
    }  
}  
  
class LaZBoy extends Recliner {  
    void prnt() {  
        System.out.println("I'm a lazboy");  
    }  
}
```



What is the output?

```
Chair cha;  
cha = new LaZBoy();  
cha.prnt();
```

```
Furniture furn;  
furn = new Recliner();  
furn.prnt();
```

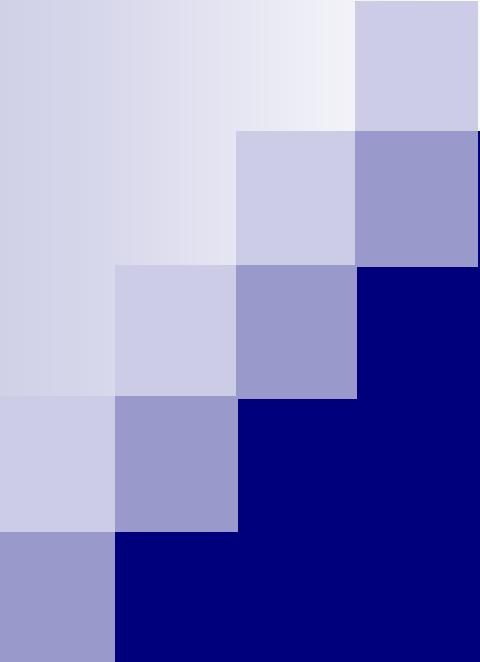
```
Furniture furn;  
furn = new LaZBoy();  
furn.prnt();
```

Quiz: Members use compile-time binding

```
class Base{  
    int X=99;  
    public void prnt(){  
        System.out.println("Base");  
    }  
}  
  
class Rtype extends Base{  
    int X=-1;  
    public void prnt(){  
        System.out.println("Rtype");  
    }  
}
```

What is the output?

```
Base b=new Rtype();  
System.out.println(b.X);  
b.prnt();
```



6 - Interfaces

Computer Science Department
California State University, Sacramento

Overview

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

Interface (Java) - Definition

An interface in the Java programming language is an abstract type that is used to specify a behavior that classes must implement. They are similar to protocols.

Interfaces are declared using the **interface** keyword, and may only contain method signature and constant declarations (variable declarations that are declared to be both static and final).

Source: [https://en.wikipedia.org/wiki/Interface_\(Java\)](https://en.wikipedia.org/wiki/Interface_(Java))

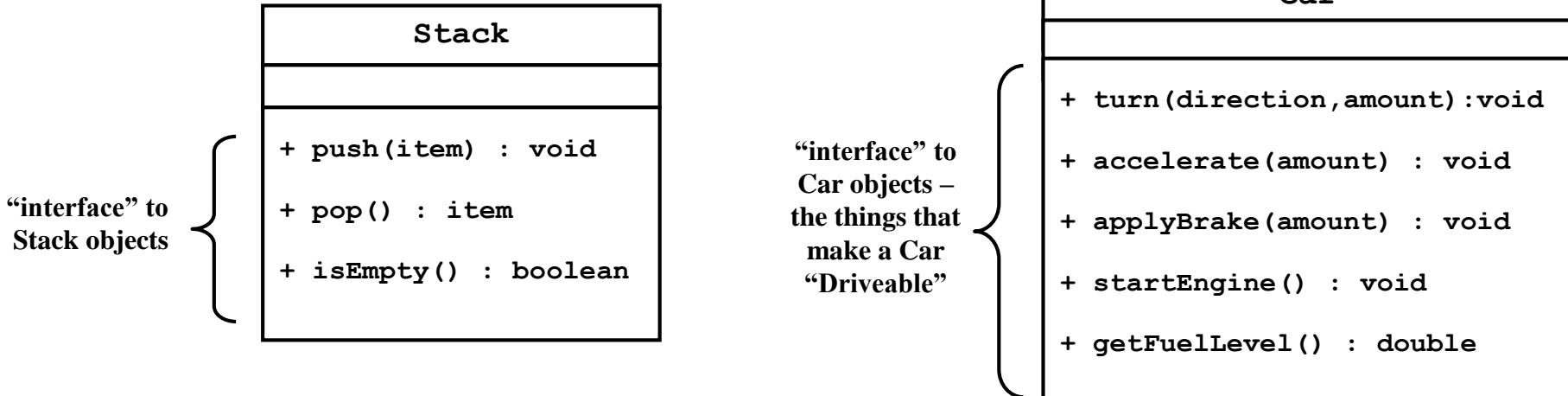
It is basically a **contract** or a promise the class has to make.

All methods of an Interface do not contain implementation (method bodies) as of all versions below Java 8.

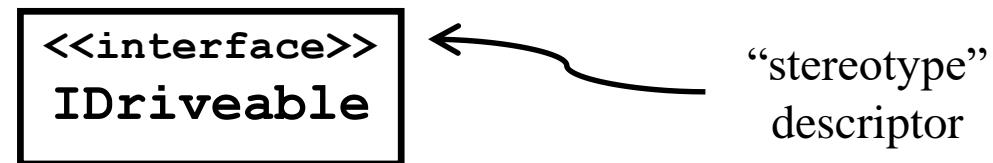
CLASS INTERFACES

Every class definition creates an “interface”

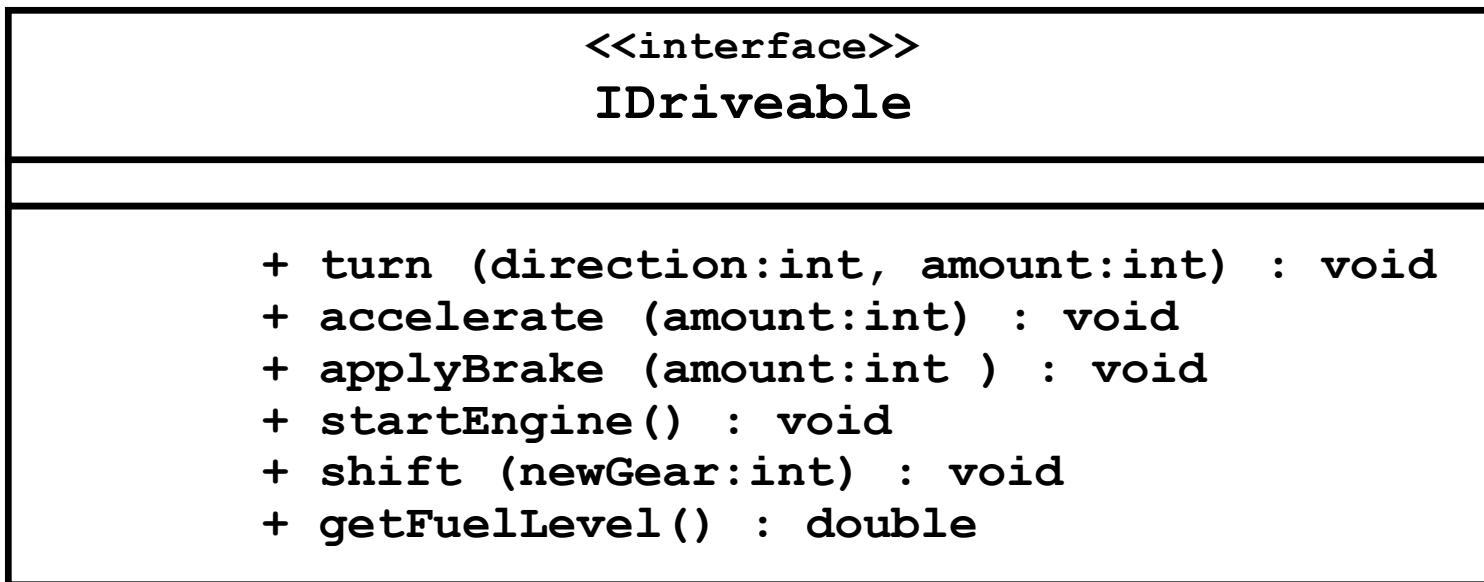
- The exposed (non-private) parts of an object



UML Interface Notation

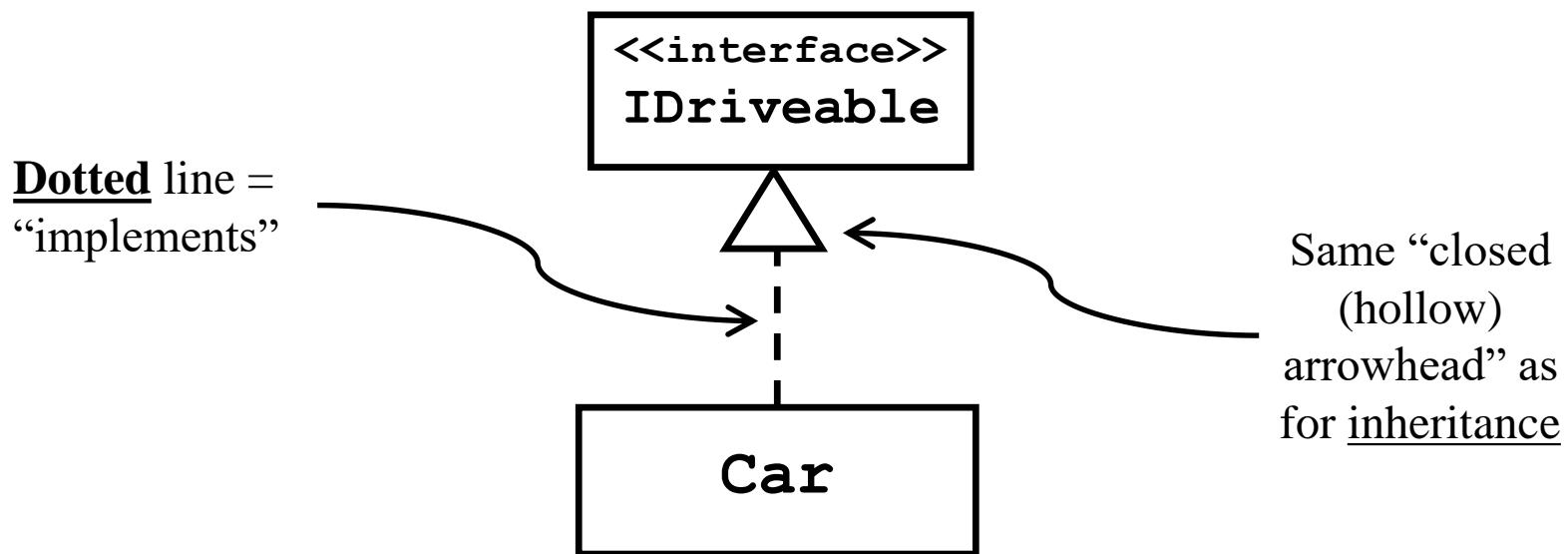


or



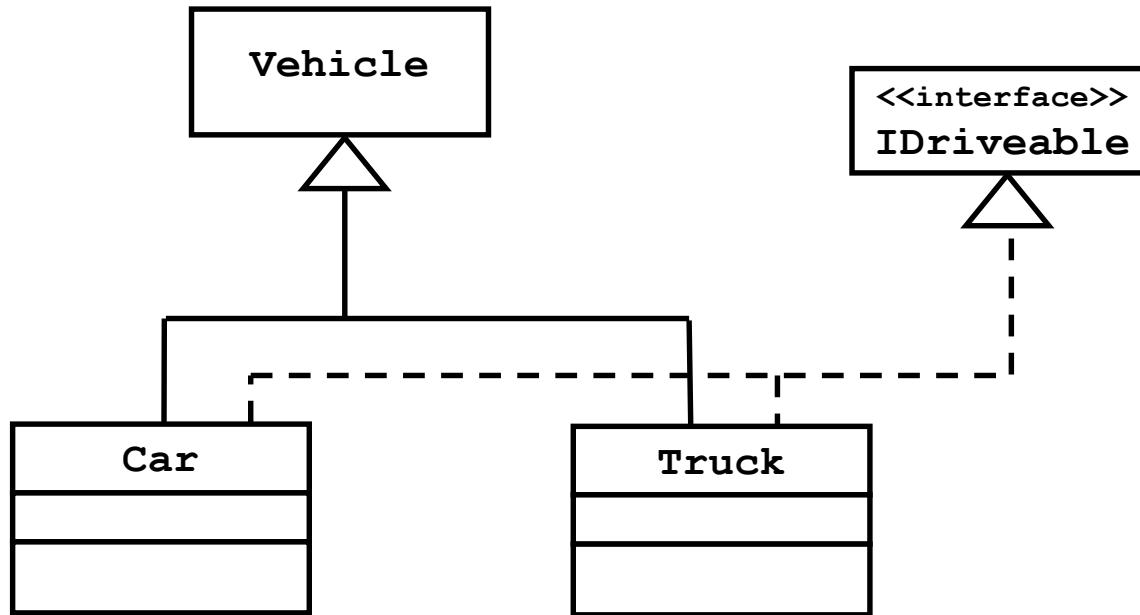
UML Interface Notation (cont.)

- 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

- o 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 :

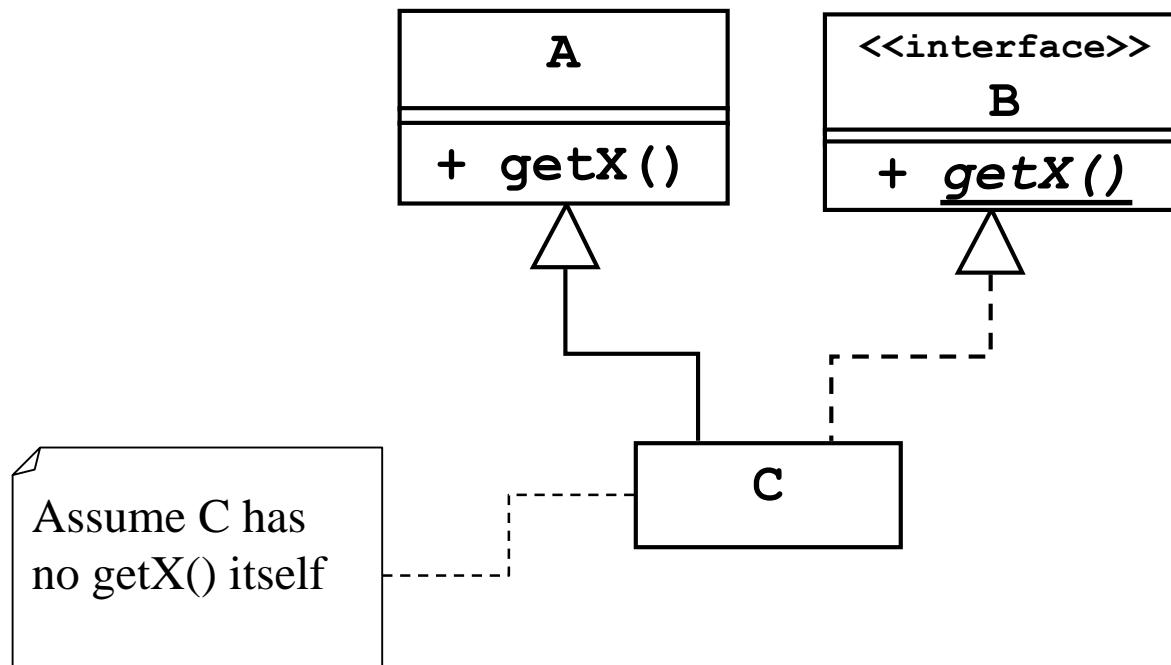
```
class IDriveable {  
public:  
    virtual void turn (int direction, int amount) = 0 ;  
    virtual void accelerate (int amount) = 0 ;  
    ...  
};
```

- “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

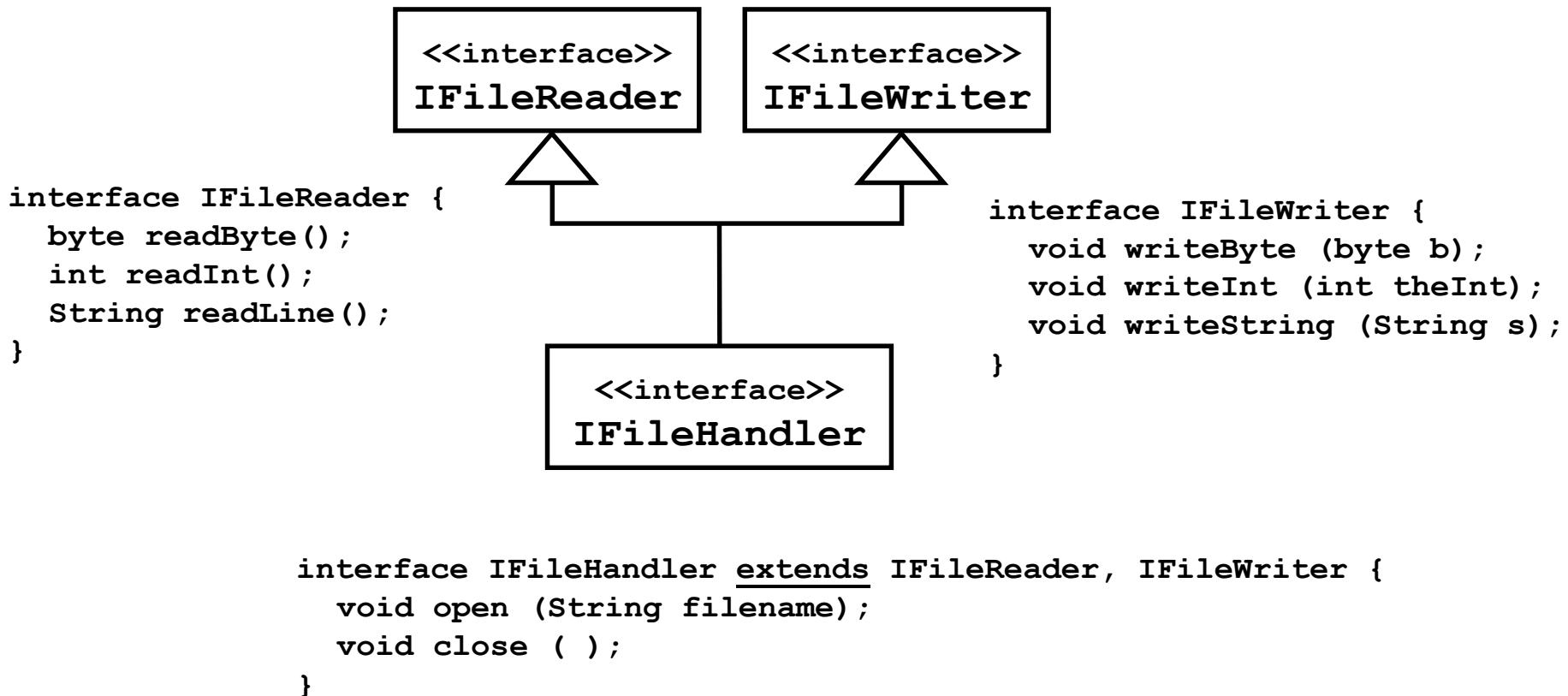
Examples:

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

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

Interface Hierarchies

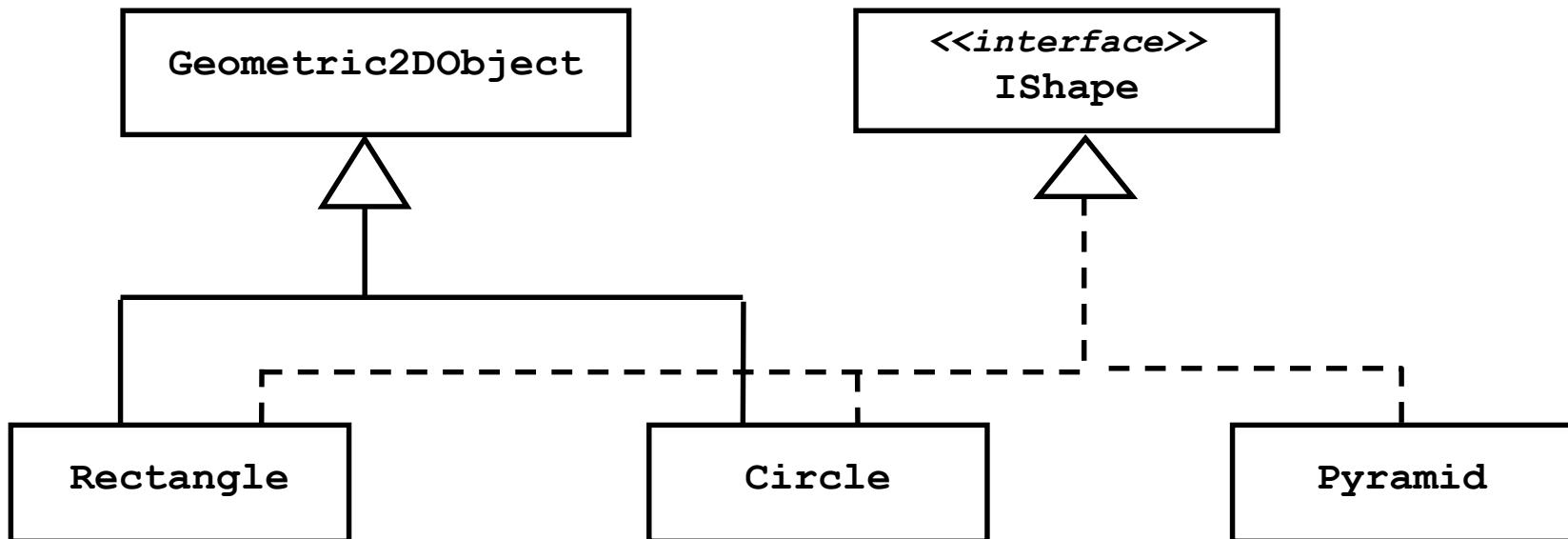
Interfaces can extend other interfaces



Interface Subtypes

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

- o A Circle “IS-A” Geometric2DObject
- o 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

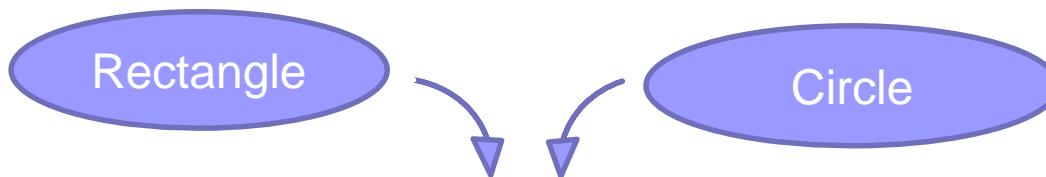
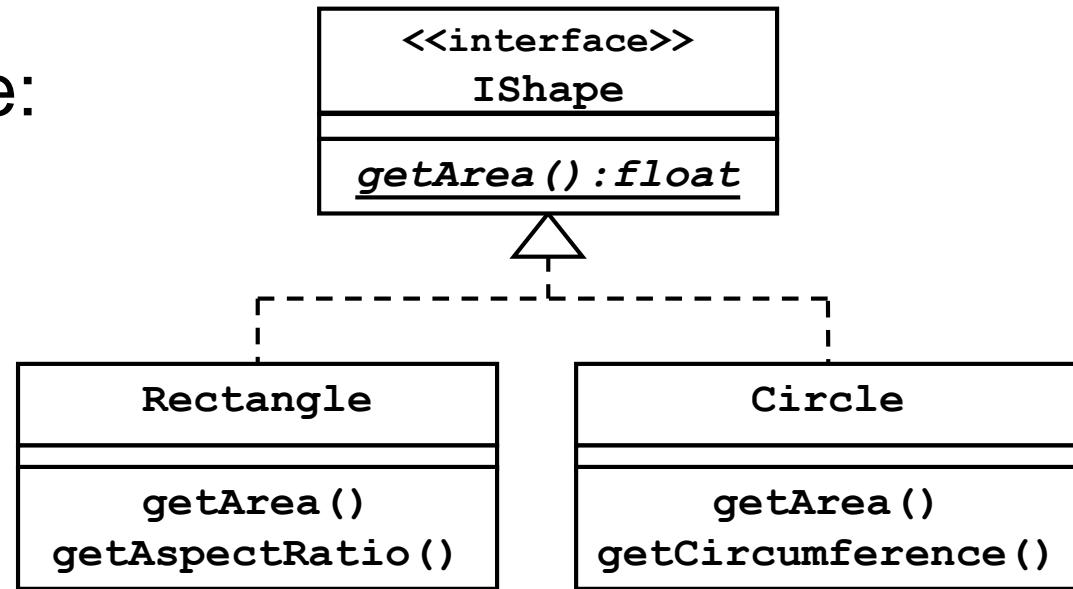
- Actual type = What was it created from (never changes)

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

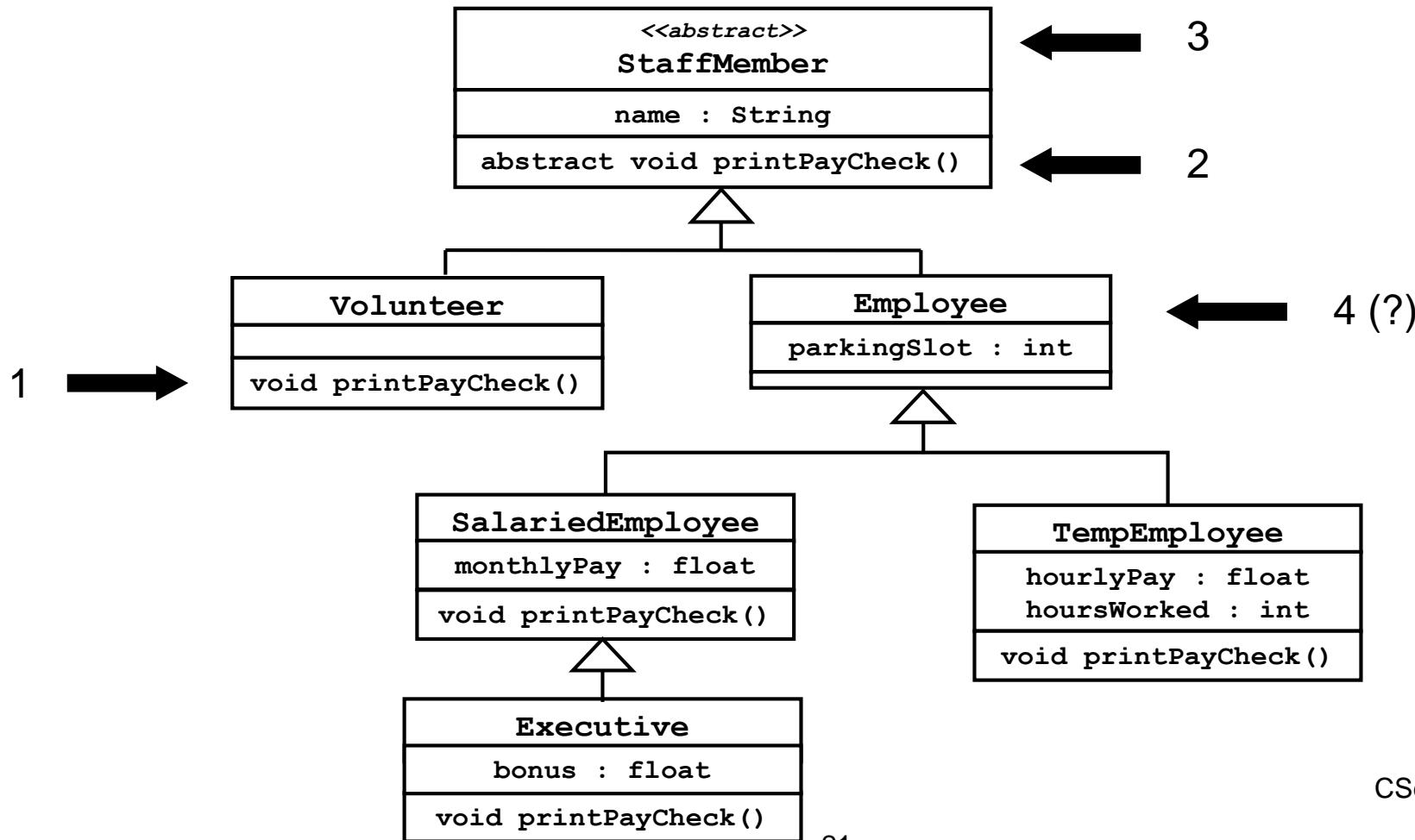
- Suppose we have:



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

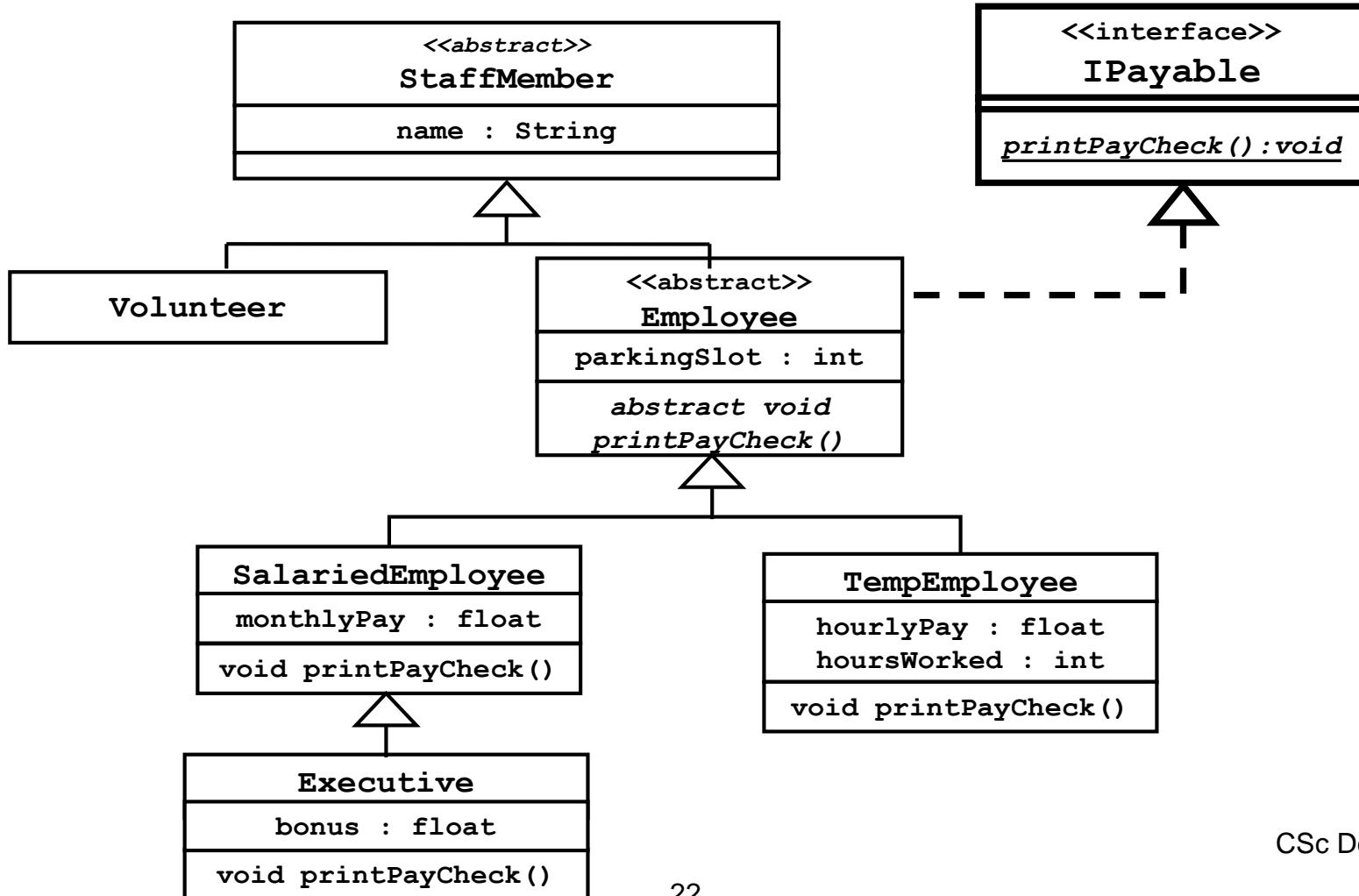
Recall: Polymorphic Safety

Ideally, every class should know how to deal with “printPayCheck” messages:



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() ;  
}
```

Increase Polymorphism

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();  
        }  
}
```

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*:

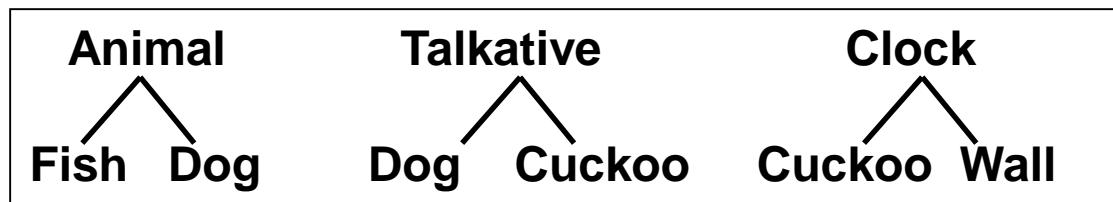
```
class CuckooClock extends Clock implements ITalkative {  
    void talk() {  
        System.out.println("Cuckoo! Cuckoo!");  
    }  
}  
  
class Interrogator {  
    static void makeItTalk(ITalkative subject) {  
        subject.talk();  
    }  
}
```

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:

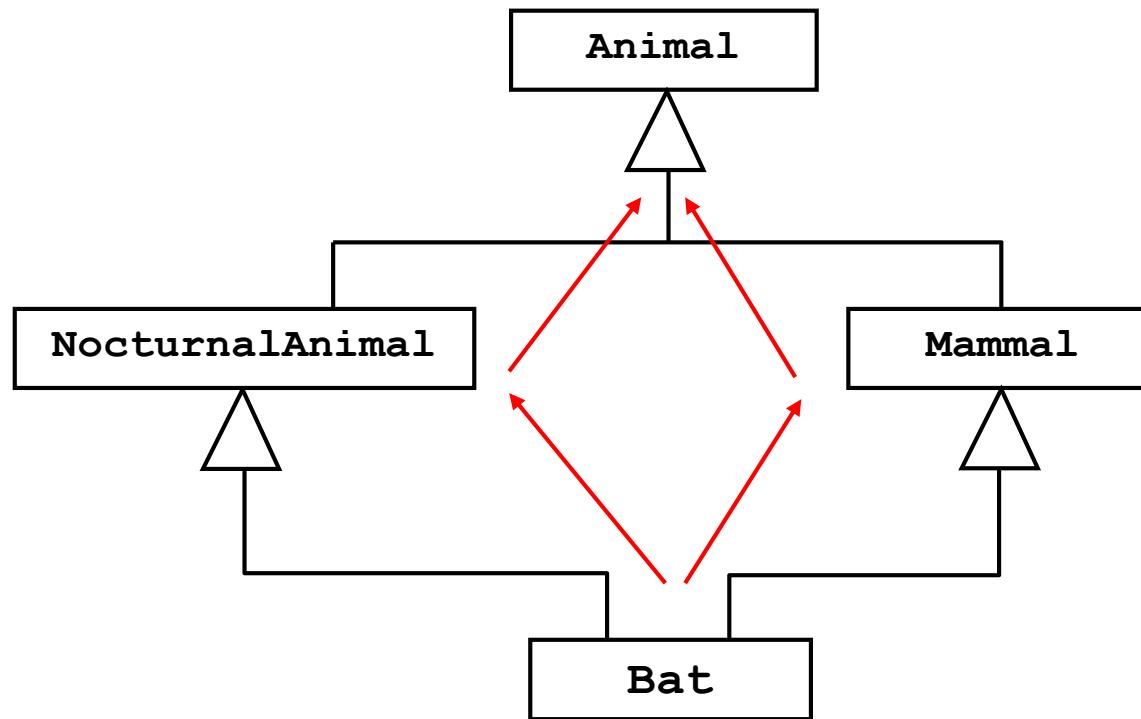
- There is a clear *inheritance hierarchy* to be defined (e.g. “kinds of animals”)
- We need non-public, non-static, or non-final fields OR private or protected methods
 - Define methods can access/modified state of objects
- Before Java 8:
 - There are at least some concrete methods shared between subclasses (share code among related classes!)
 - 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

Recall: Multiple Inheritance

- C++ allows multiple inheritance:^(cont.)

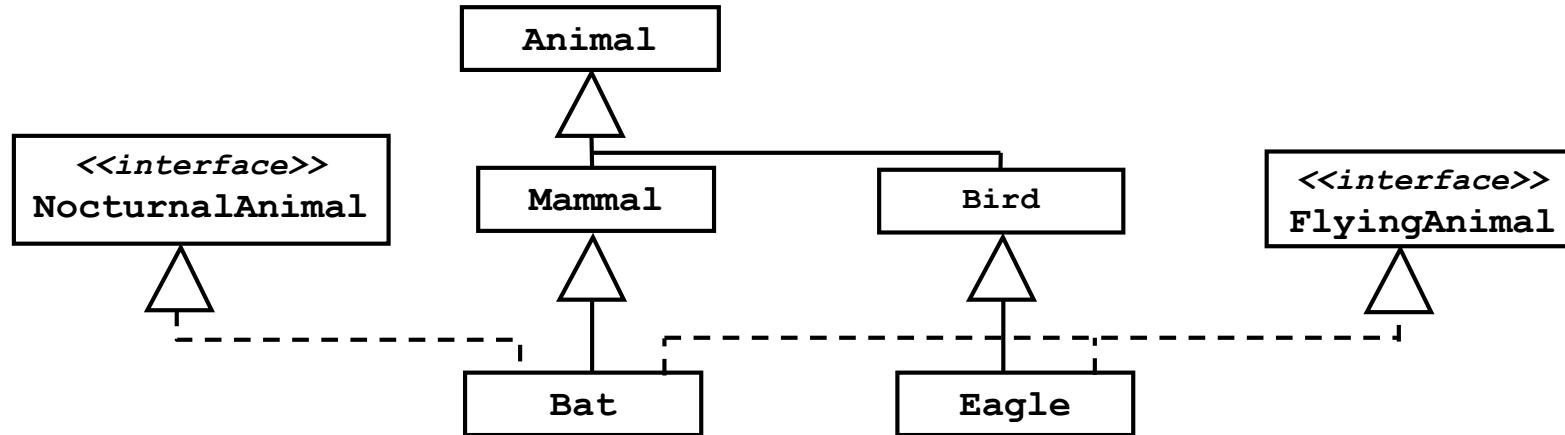
```
class Animal{...};  
  
class Mammal : Animal {  
    public : void sleep() {...} ;  
    ...  
};  
  
class NocturnalAnimal : Animal {  
    public : void sleep() {...} ;  
    ...  
};  
  
class Bat : Mammal, NocturnalAnimal {...};
```

- Programmer must disambiguate references:

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

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 {...}
```

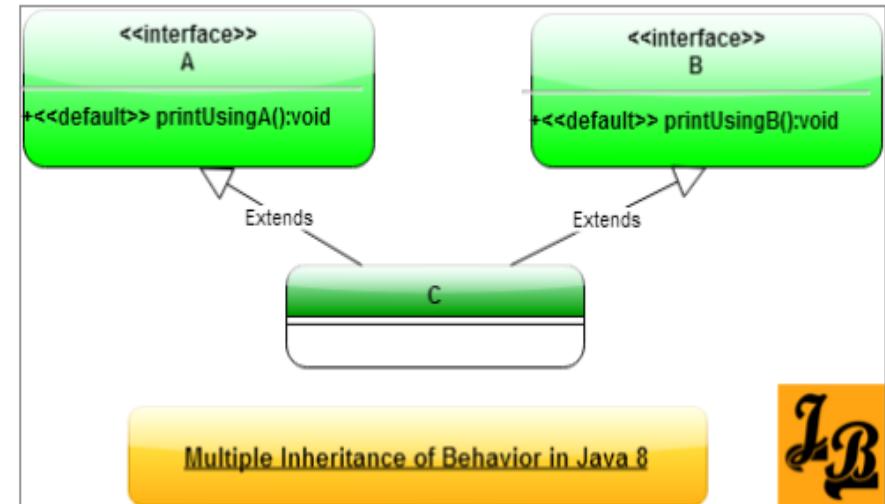
Does Java support multiple inheritance?

- o Of interfaces – Yes
- o Of implementations – No (before Java 8)

Backup Slide on Multiple Inheritance

Java 8 code showing multiple inheritance for interfaces A,B & class C

```
//A.java
public interface A{
    public default void printUsingA(){
        System.out.println("Print from A");
    }
}
//B.java
public interface B{
    public default void printUsingB(){
        System.out.println("Print from B");
    }
}
//C.java
public class C implements A,B{
    public static void main(String args[]){
        C cObj=new C();
        cObj.printUsingA();
        cObj.printUsingB();
    }
}
```



OUTPUT of the above code

```
Print from A
Print from B
```

Attendance Quiz - Practice

```
public class Eye extends Mouth {  
    public void method1() {  
        S.o.println("Eye 1");  
        super.method1();  
    }  
}  
public class Mouth {  
    public void method1() {  
        S.o.println("Mouth 1");  
    }  
    public void method2() {  
        S.o.println("Mouth 2");  
        method1();  
    }  
}  
public class Nose extends Eye {  
    public void method1() {  
        S.o.println("Nose 1");  
    }  
    public void method3() {  
        S.o.println("Nose 3");  
    }  
}  
public class Ear extends Eye {  
    public void method2() {  
        S.o.println("Ear 2");  
    }  
    public void method3() {  
        S.o.println("Ear 3");  
    }  
}
```

The following variables are defined:

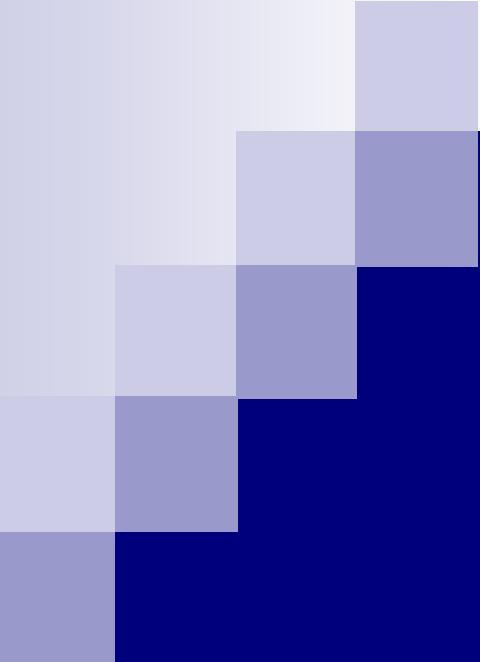
```
Mouth var1 = new Nose();  
Ear var2 = new Ear();  
Mouth var3 = new Eye();  
Object var4 = new Mouth();  
Eye var5 = new Nose();  
Mouth var6 = new Ear();
```

<u>Statement</u>	<u>Output</u>
------------------	---------------

var5.method2(); _____

var6.method2(); _____

var1.method3(); _____



7 - Design Patterns (Part II)

Computer Science Department
California State University, Sacramento

Overview

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

<i>Composite</i>	<i>Singleton</i>
<i>Iterator</i>	<i>Observer</i>
<i>Strategy</i>	<i>Command</i>
<i>Proxy</i>	<i>Factory Method</i>
- **MVC Architecture**

Midterm Schedule

Section 2 and 4:
March 12th , 2019

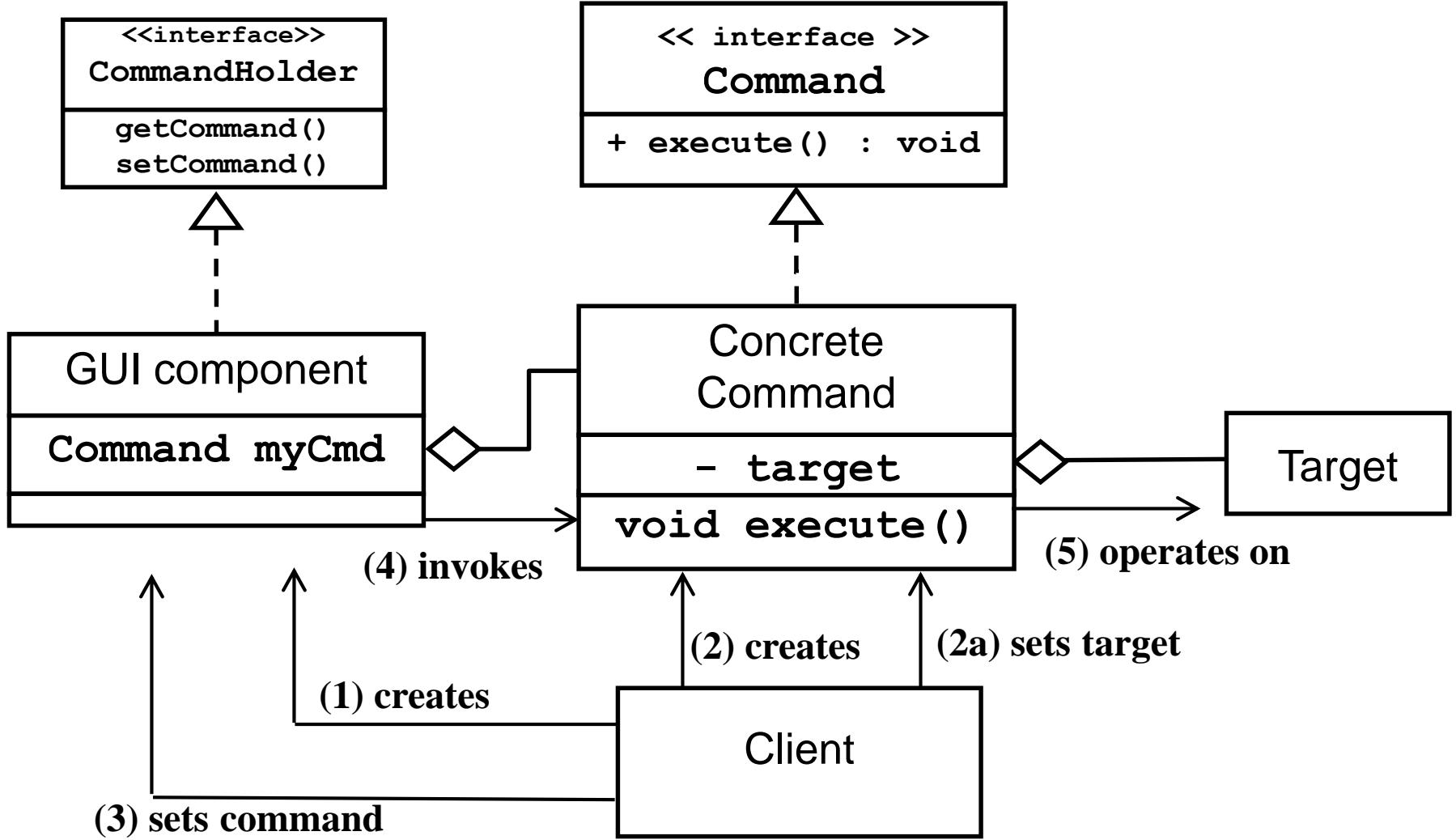
Part II – Design Pattern

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 subclass 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]
```

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:**
 - (1) buttons w/ **setCommand()**, (2) title bar area items w/ **Toolbar's addCommandToXXX()** methods, (3) 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 buttons 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 command 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...");  
    }  
}
```

Example: Command Design Pattern - Set Target

```
6 public class AddAsteroidCommand extends Command {  
7     //~~~~~ F I E L D S ~~~~~/  
8  
9  
10  
11     private GameWorld gw;    //Reference to a Game World  
12     //~~~~~ C O N S T R U C T O R S ~~~~~/  
13  
14     /* There is only one constructor.  
15     */  
16     public AddAsteroidCommand( GameWorld gw ){  
17         super( "Add Asteroid" );  
18         this.gw = gw;  
19     }  
20     //~~~~~ M E T H O D S ~~~~~/  
21  
22     //There is only one method to override the action performed  
23     @Override  
24     public void actionPerformed( ActionEvent e ){  
25         gw.addAsteroid();  
26         System.out.println("Add Asteroid.");  
27     }  
28  
29 }  
30  
31 }
```

(Note: assignment

Organized commands in package for ease of accessment

```
com.mycompany.a4.commands
├── AboutCommand.java
├── AddAsteroidCommand.java
├── AddShipCommand.java
├── AddSpaceStationCommand.java
├── DecreaseShipSpeedCommand.java
├── FireShipMissileCommand.java
├── FireShipPlasmaWaveCommand.java
├── IncreaseShipSpeedCommand.java
├── JumpShipThroughHyperspaceCommand.java
├── NewCommand.java
├── QuitGameCommand.java
├── RefuelSelectedObjectsCommand.java
├── RestartGameCommand.java
├── SaveCommand.java
├── ToggleDrawControlPointsCommand.java
├── TogglePauseCommand.java
├── ToggleSoundsCommand.java
├── TurnShipLeftCommnad.java
├── TurnShipRightCommand.java
└── UndoCommand.java
```

A file tree diagram showing the contents of the 'com.mycompany.a4.commands' package. The package is expanded, revealing 18 Java source files. Each file is represented by a small icon followed by its name in blue text. The files are: AboutCommand.java, AddAsteroidCommand.java, AddShipCommand.java, AddSpaceStationCommand.java, DecreaseShipSpeedCommand.java, FireShipMissileCommand.java, FireShipPlasmaWaveCommand.java, IncreaseShipSpeedCommand.java, JumpShipThroughHyperspaceCommand.java, NewCommand.java, QuitGameCommand.java, RefuelSelectedObjectsCommand.java, RestartGameCommand.java, SaveCommand.java, ToggleDrawControlPointsCommand.java, TogglePauseCommand.java, ToggleSoundsCommand.java, TurnShipLeftCommnad.java, TurnShipRightCommand.java, and UndoCommand.java.

Example: Command Design Pattern

Arrows and Space bar (Cont)

The second input mechanism will use CN1 *Key Binding* concepts so that the *left arrow*, *right arrow*, *up arrow*, and *down arrow* keys invoke command objects corresponding to the code previously executed when the “l”, “r”, “i”, and “d” keys (for changing the ship direction and speed) were entered, respectively. Note that this means that whenever an arrow key is pressed, the program will *immediately* invoke the corresponding action (no “Enter” key press is required). The program is also to use key bindings to bind the SPACE bar to the “fire missile” command and the “j” key to the “jump through hyperspace” command. If you want, you may also use key bindings to map any of the other command keys from A1, but only the ones listed above are required.

```
//===== Binding play-mode specific key commands =====
// Binding up arrow to increase ship speed (up arrow key code = -91)
addKeyListener(-91, myIncreaseShipSpeedCmd);
// Binding down arrow to decrease ship speed (down arrow key code = 92)
addKeyListener(-92, myDecreaseShipSpeedCmd);
// Binding left arrow to turn the ship left (left arrow key code = -93)
addKeyListener(-93, myTurnShipLeftCmd);
// Binding right arrow to turn the ship right (right arrow key code = -94)
addKeyListener(-94, myTurnShipRightCmd);
// Binding the space bar to fire a ship missile (space bar key code = -90) (this also binds the enter key)
addKeyListener(-90, myFireShipMissileCmd);

addKeyListener(44, mGCmdl); // MissileLauncher Left turn
addKeyListener(46, mGCmdr); // MissileLauncher Right turn
```

The Strategy Pattern

Motivation

- o A variety of algorithms exists to perform a particular operation
- o 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?

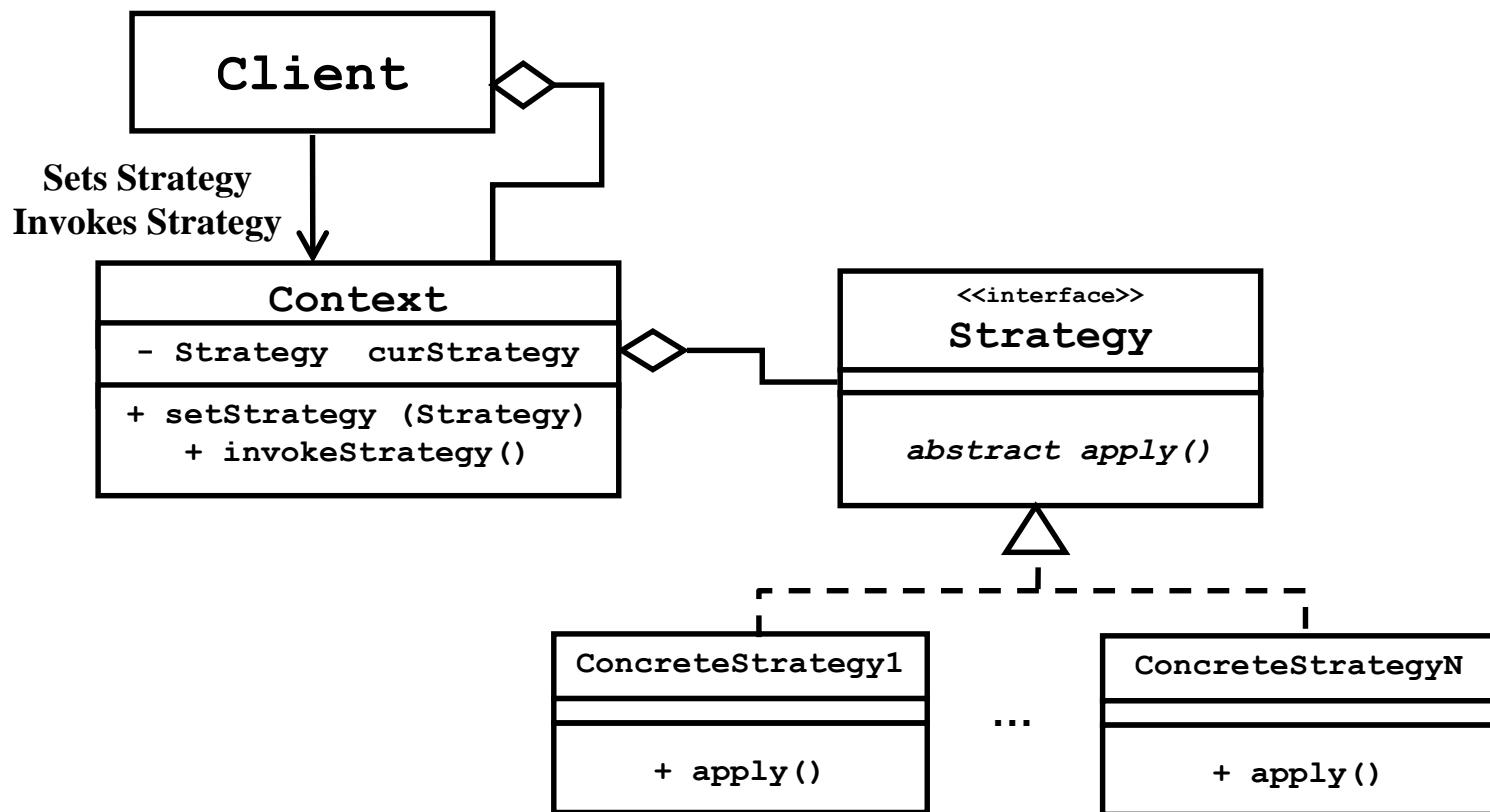
Changing or adding a plan requires changing the client!

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

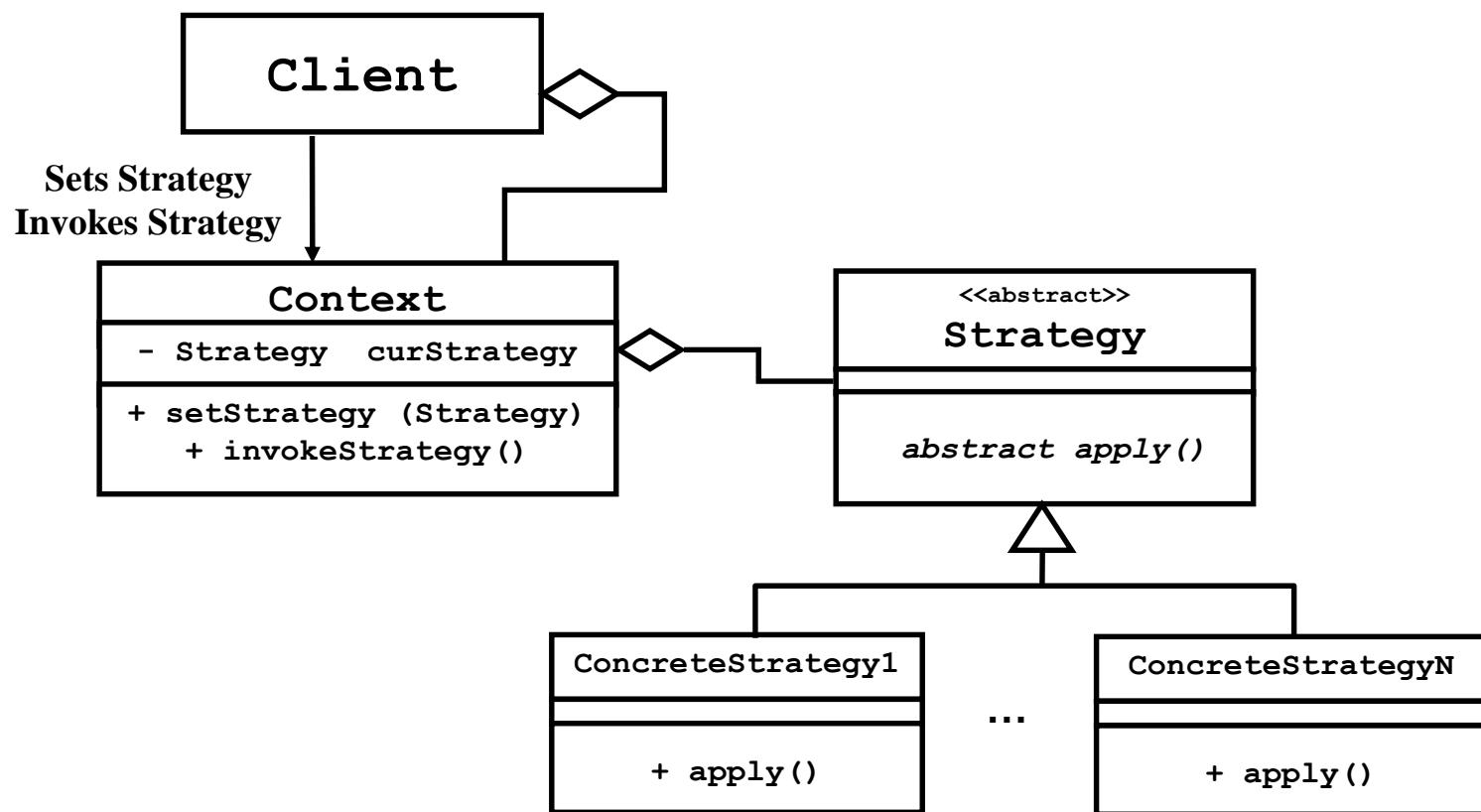
Strategy Pattern Organization

- Using Interfaces

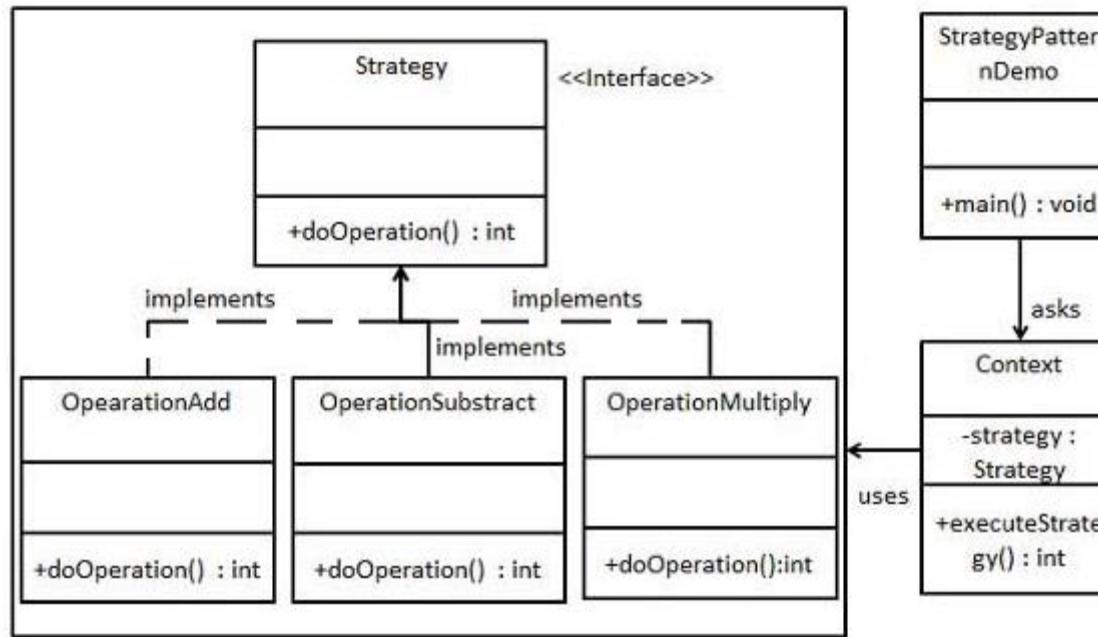


Strategy Pattern Organization (cont.)

- Using subclassing



Example: StrategyPatternDemo (Using Interface)



StrategyPatternDemo will use Context and strategy objects to demonstrate change in Context behavior based on strategy it deploys or uses.

Source: https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm

Example: StrategyPatternDemo

(Cont)

Step 1: Create an interface

```
public interface Strategy {  
    public int doOperation(int  
        num1, int num2);  
}
```

Step 2: Create concrete classes implementing the same interface.

```
public class OperationAdd implements Strategy{  
    @Override  
    public int doOperation(int num1, int num2) {  
        return num1 + num2;  
    }  
}
```

```
public class OperationSubstract implements  
Strategy{  
    @Override  
    public int doOperation(int num1, int num2) {  
        return num1 - num2;  
    }  
}
```

```
public class OperationMultiply implements  
Strategy{  
    @Override  
    public int doOperation(int num1, int num2) {  
        return num1 * num2;  
    }  
}
```

Example: StrategyPatternDemo (Cont)

Step 3: Create context class

```
public class Context {  
    private Strategy strategy;  
  
    public Context(Strategy  
strategy){  
        this.strategy = strategy;  
    }  
  
    public int executeStrategy(int  
num1, int num2){  
        return  
strategy.doOperation(num1, num2);  
    }  
}
```

Step 5: Verify result:

```
10 + 5 = 15  
10 - 5 = 5  
10 * 5 = 50
```

Step 4: Use the Context to see change in behavior when it changes its Strategy..

```
public class StrategyPatternDemo {  
    public static void main(String[] args) {  
        Context context = new Context(new OperationAdd());  
  
        System.out.println("10 + 5 = " +  
context.executeStrategy(10, 5));  
  
        context = new Context(new OperationSubtract());  
        System.out.println("10 - 5 = " +  
context.executeStrategy(10, 5));  
  
        context = new Context(new OperationMultiply());  
        System.out.println("10 * 5 = " +  
context.executeStrategy(10, 5));  
    }  
}
```

Example: 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)
```

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

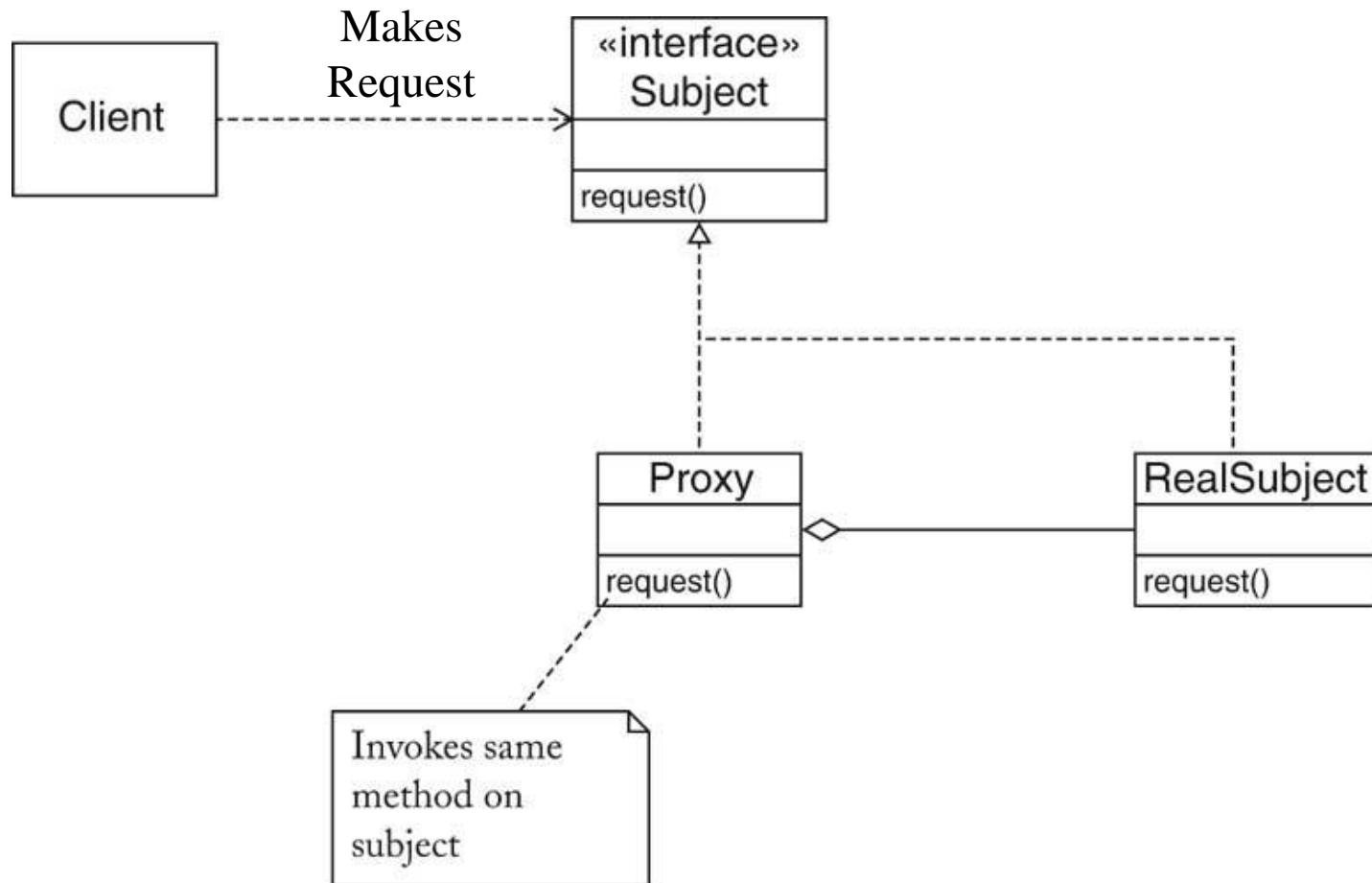
The Proxy Pattern

- 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

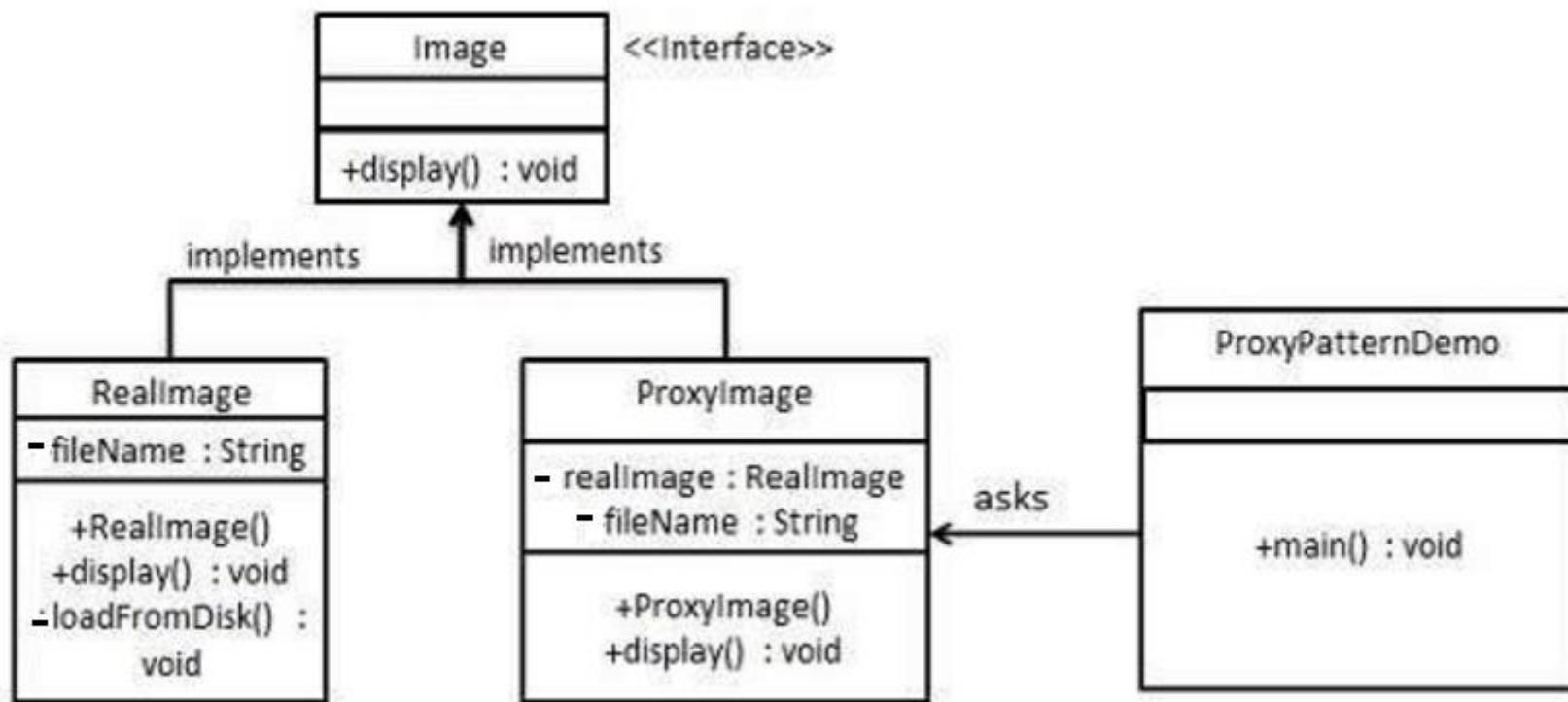
- **Protection Proxy** – controls access
- **Virtual Proxy** – acts as a stand-in
- **Remote Proxy** – local stand-in for object in another address space
- **This type of design pattern comes under structural pattern.**

Proxy Pattern Organization



Proxy Example 1

ProxyImage is a proxy class to reduce memory footprint of ReallImage object loading. ProxyPatternDemo, our demo class, will use ProxyImage to get an Image object to load and display as it needs.

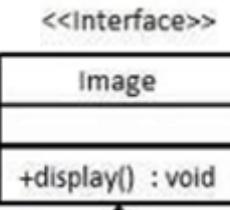


Source: https://www.tutorialspoint.com/design_pattern/proxy_pattern.htm

Proxy Example 1 (Cont)

Step 1: Create an interface

```
public interface Image {
    void display();
}
```

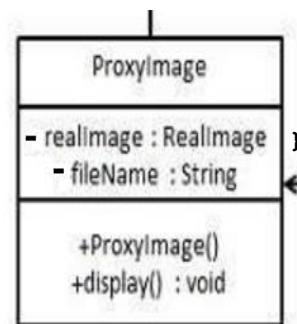


Step 2: Create concrete classes implementing the same interface.

```
public class ProxyImage implements Image{
    private RealImage realImage;
    private String fileName;

    public ProxyImage(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void display() {
        if(realImage == null){
            realImage = new RealImage(fileName);
        }
        realImage.display();
    }
}
```



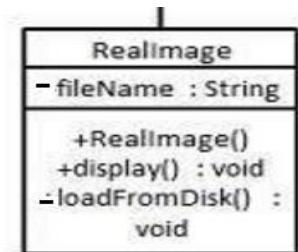
Step 2: Create concrete classes implementing the same interface.

```
public class RealImage implements Image {
    private String fileName;

    public RealImage(String fileName) {
        this.fileName = fileName;
        loadFromDisk(fileName);
    }

    @Override
    public void display() {
        System.out.println("Displaying " + fileName);
    }

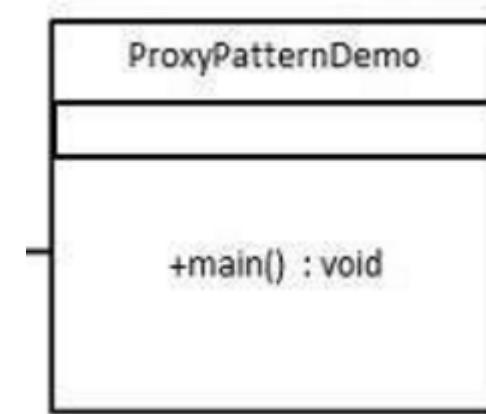
    private void loadFromDisk(String fileName) {
        System.out.println("Loading " + fileName);
    }
}
```



Proxy Example 1 (Cont)

Step 3: Use the `ProxyImage` to get object of `RealImage` class when required..

```
public class ProxyPatternDemo {  
  
    public static void main(String[] args) {  
        Image image = new ProxyImage("test_10mb.jpg");  
  
        //image will be loaded from disk  
        image.display();  
        System.out.println("");  
  
        //image will not be loaded from disk  
        image.display();  
    }  
}
```



Step 4: Verify result.

```
Loading test_10mb.jpg  
Displaying test_10mb.jpg  
  
Displaying test_10mb.jpg
```

Proxy Example 2

```
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 2 (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**
 - We create object without exposing the creation logic to the client and refer to newly created object using a common interface.
 - 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());

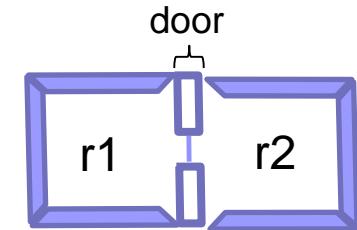
        r2.setSide(NORTH, new Wall());
        r2.setSide(EAST, new Wall());
        r2.setSide(SOUTH, new Wall());
        r2.setSide(WEST, theDoor);

        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()...)
}

```



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) {

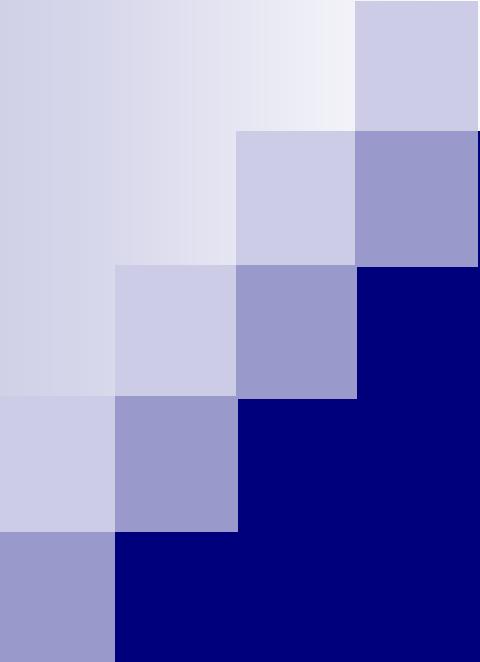
        //create the spell necessary to enter the enchanted room
        Spell spell = makeSpell() ;

        //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() ;}
    ...
}
```



7 - Design Patterns

Computer Science Department
California State University, Sacramento

Overview

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

<i>Composite</i>	<i>Singleton</i>
<i>Iterator</i>	<i>Observer</i>
<i>Strategy</i>	<i>Command</i>
<i>Proxy</i>	<i>Factory Method</i>
- **MVC Architecture**

Definition

In software engineering, a software design pattern is a **general reusable solution** to a commonly occurring problem within a given context in software design. It is **not** a finished design that can be transformed directly into source or machine code.

Source: https://en.wikipedia.org/wiki/Software_design_pattern

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 (i.e. **Singleton**)
- **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
 - i.e. **Composite**
- **BEHAVIORAL**
 - Deal with interaction between objects
 - Encapsulate processes performed by objects
 - i.e. **Iterator**

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***

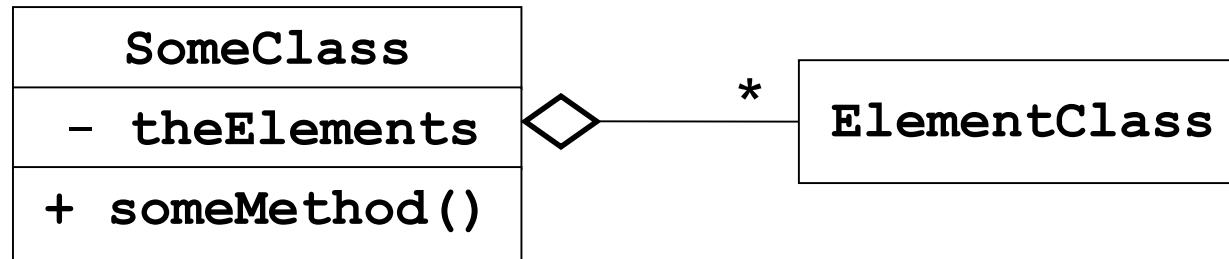
Behavioral:

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

The Iterator Pattern

- MOTIVATION

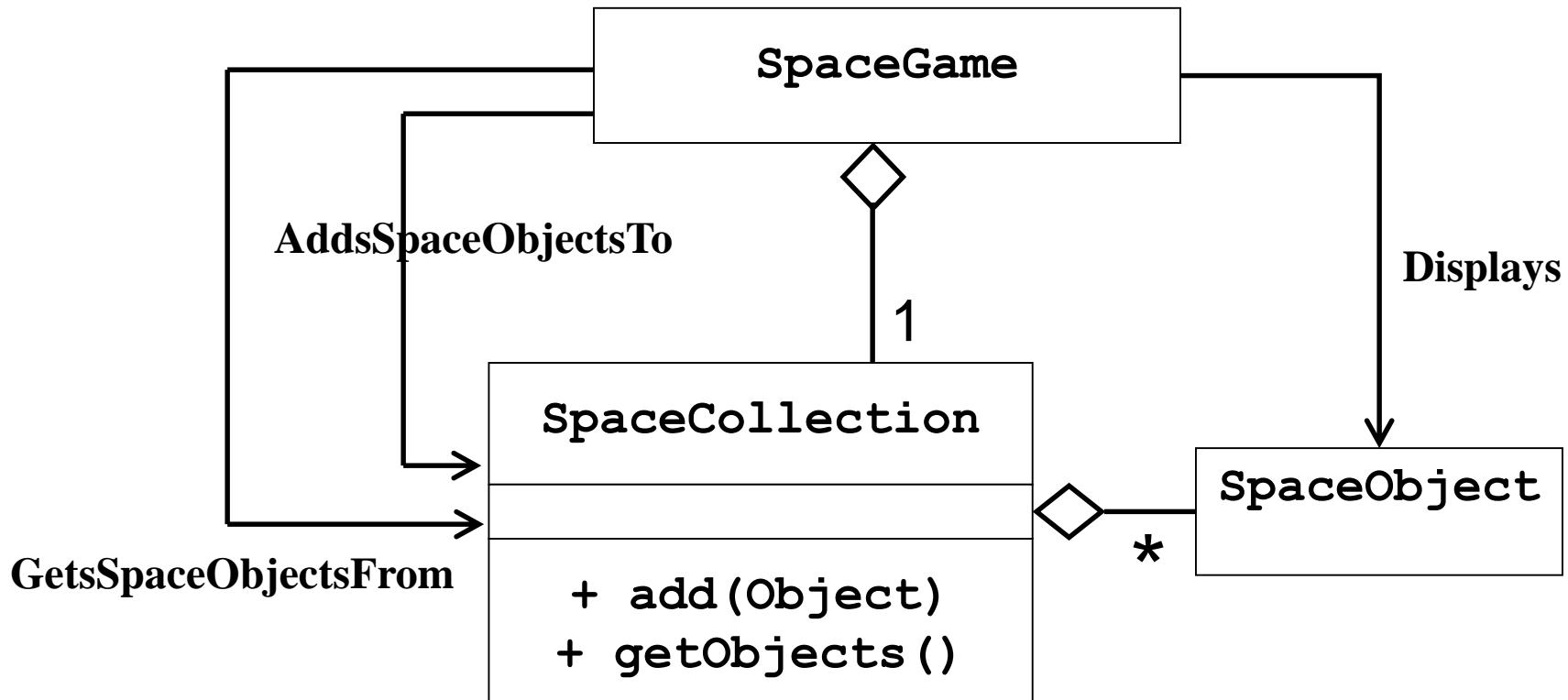
- An “aggregate” object contains “elements”
- “Clients” need to access these elements
- Aggregate shouldn’t expose internal structure



- 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

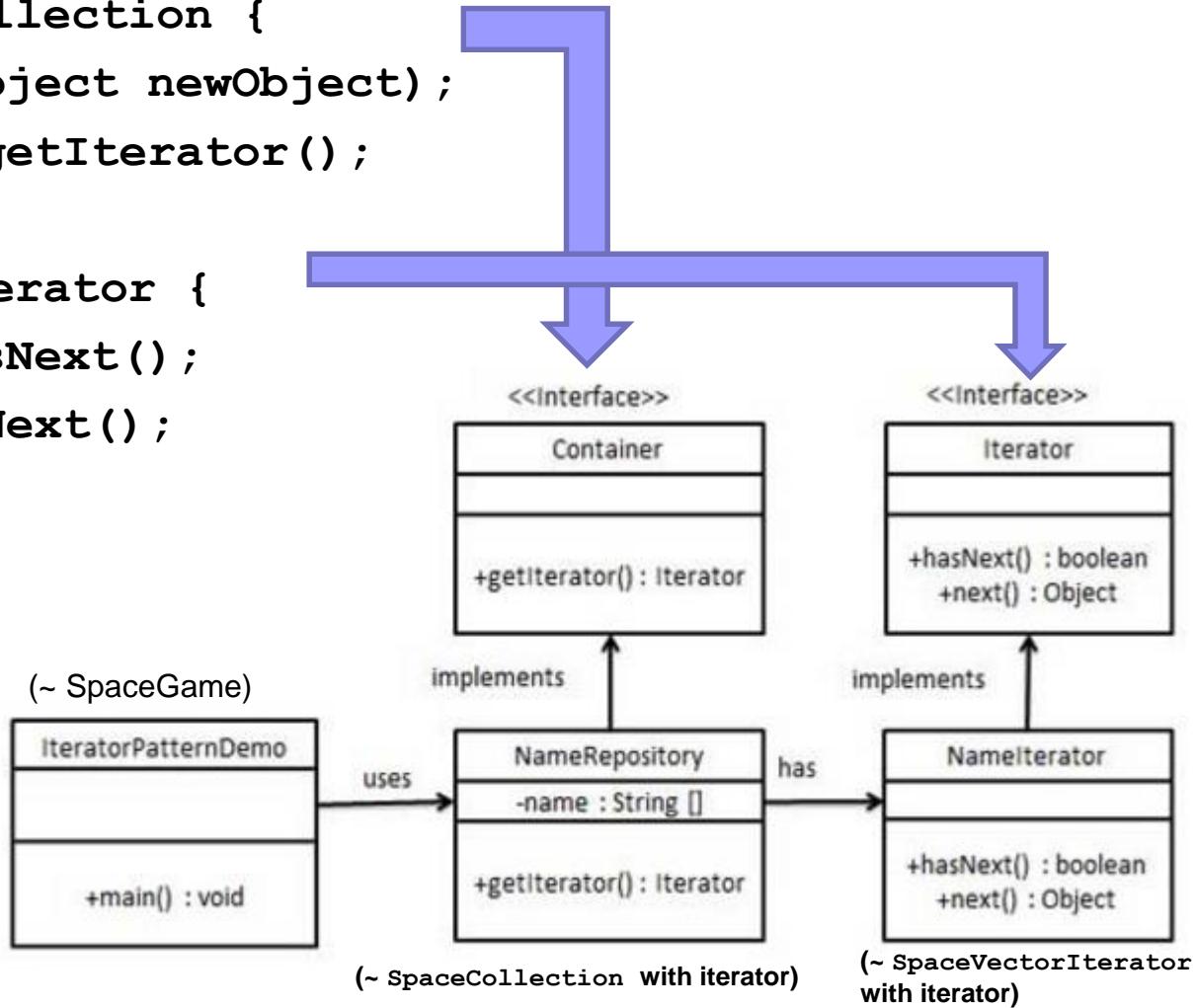
```
public interface ICollection {  
    public void add(Object newObject);  
    public IIIterator getIterator();  
}  
  
public interface IIIterator {  
    public boolean hasNext();  
    public Object getNext();  
}
```

(~ `SpaceCollection` with iterator)

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 IIIterator getIterator() {  
        return new SpaceVectorIterator ( ) ;  
    }  
  
    ...continued...
```

SpaceCollection With Iterator (cont.)

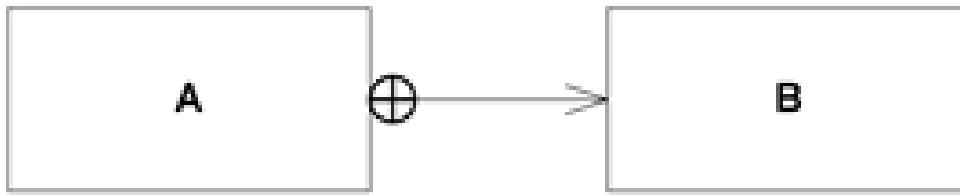
```
private class SpaceVectorIterator implements IIerator {
    private int currElementIndex;

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

Quiz:
Which method get
called here?

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());  
        }  
    }  
}
```

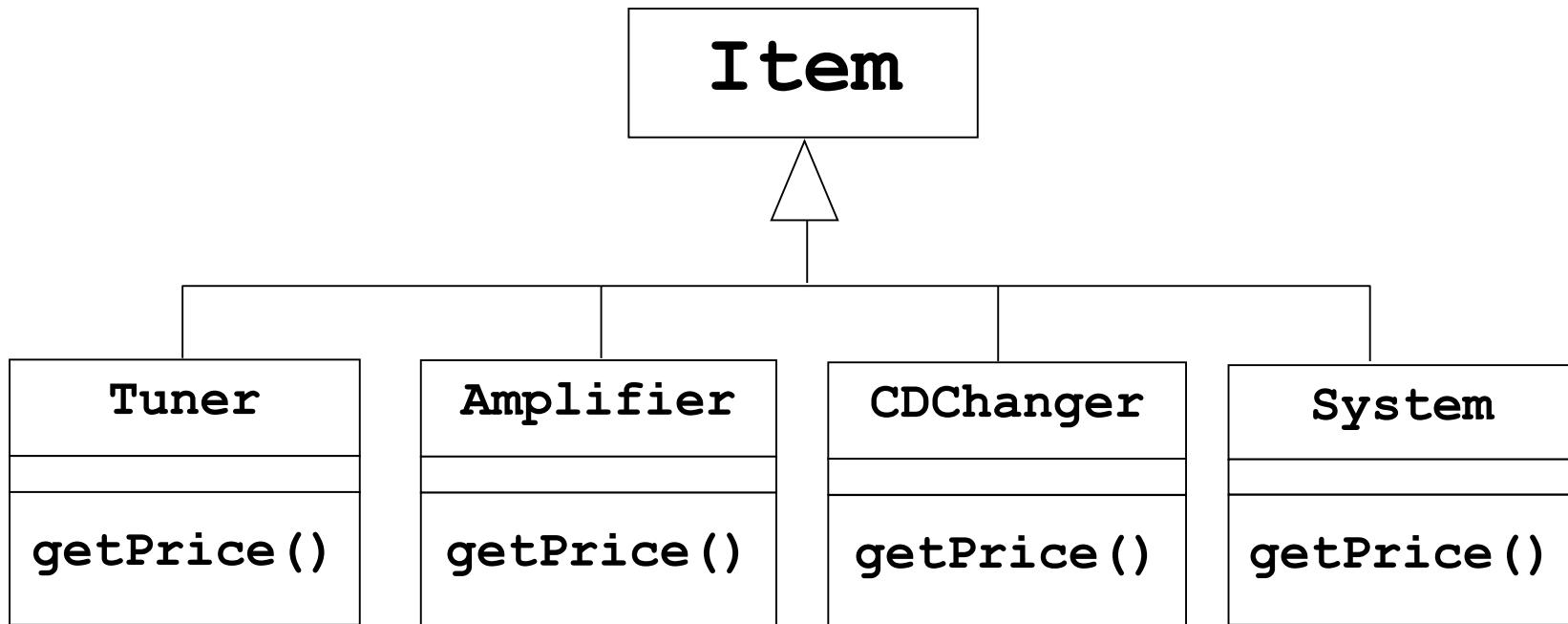
Recap: The Iterator Pattern

- This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.
- Iterator pattern falls under behavioral pattern category.

The Composite Pattern

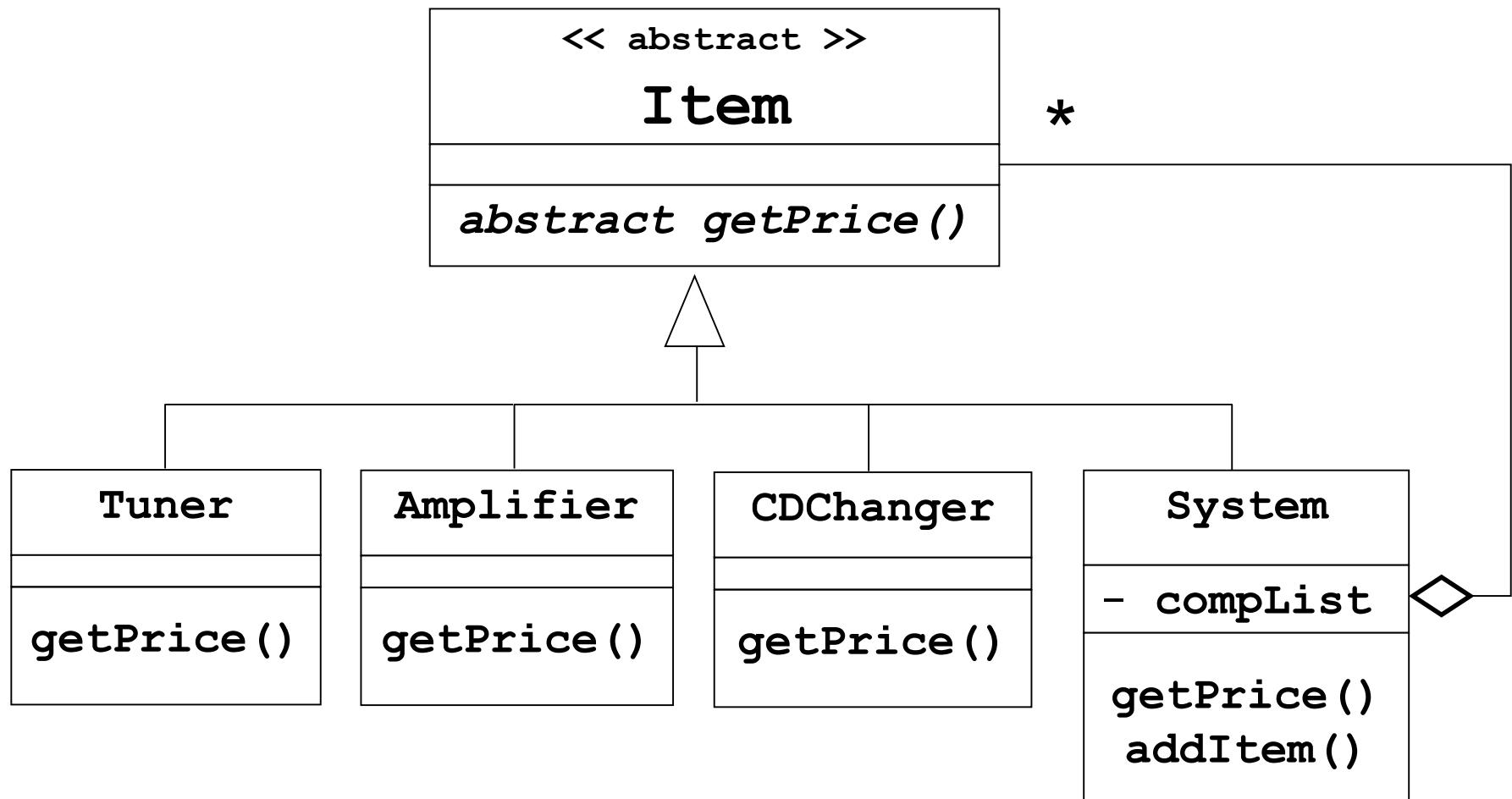
- **MOTIVATION:**
 - Objects organized in a hierarchical manner
 - Some objects are *groups* of the other objects
 - Individuals and groups need to be treated uniformly
- **Example:**
 - A store sells stereo component items:
Tuners, Amplifiers, CDChangers, etc.
 - Each item has a `getPrice()` method
 - The store also sells complete stereo systems
 - Systems also have a `getPrice()` method which returns a discounted sum of the prices.

Possible Class Organization

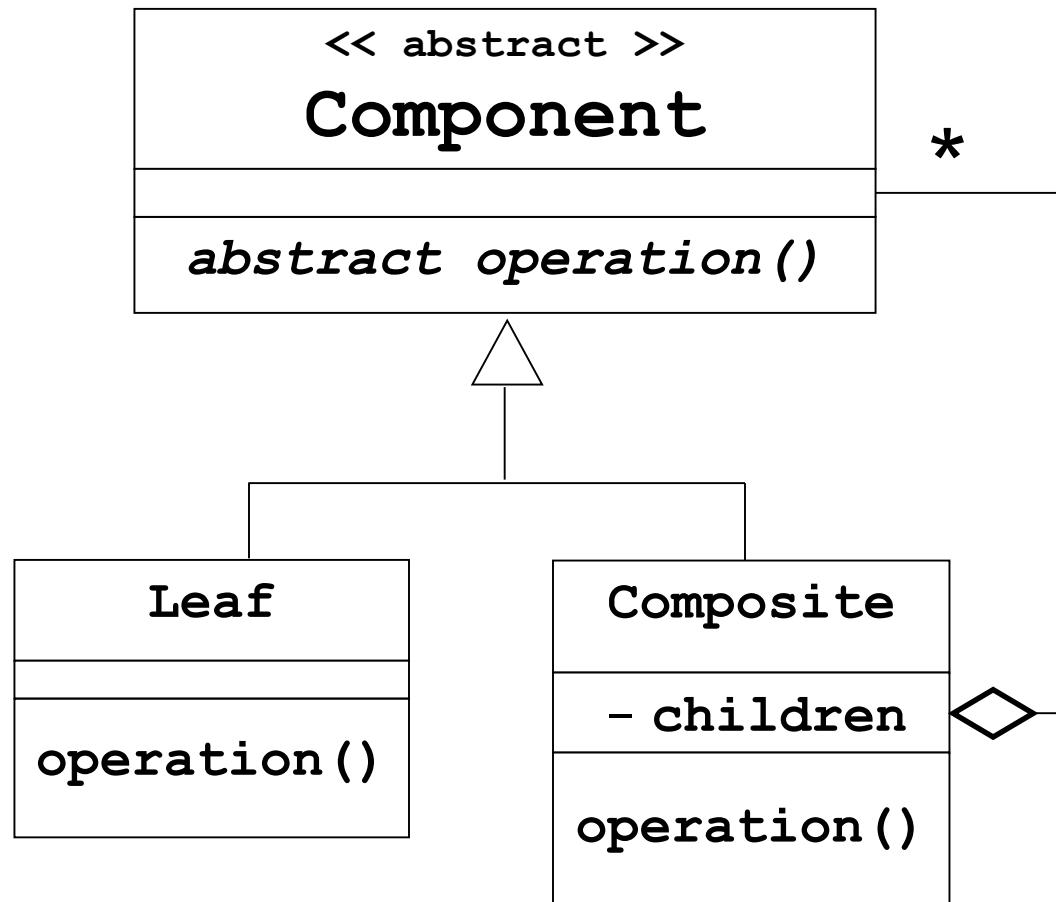


Problem ?

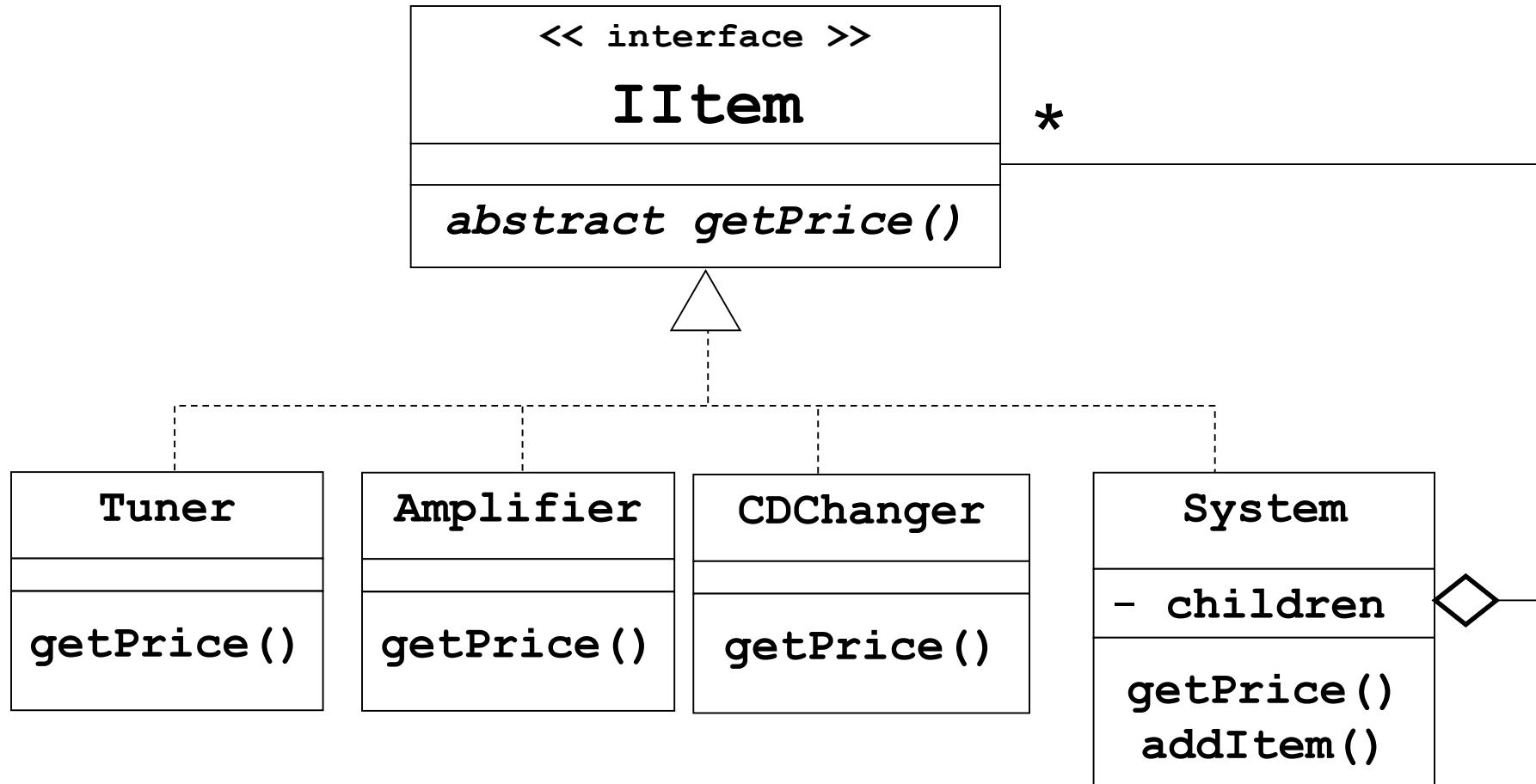
Solution



Composite Pattern Organization



Composite Specified With Interfaces



Other Examples Of Composites

- Trees
 - Internal nodes (groups) and leaves
- Arithmetic expressions
 - $\langle \text{exp} \rangle ::= \langle \text{term} \rangle \mid \langle \text{term} \rangle "+" \langle \text{exp} \rangle$
- Graphical Objects
 - Rectangles, lines, circles
 - Frames
 - can contain other graphical objects

Recap: The Composite Pattern

- Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy.
- This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects.

The Singleton Pattern

- **Definition**

- The **singleton pattern** is a software design pattern that restricts the instantiation of a class to one object.

- **Motivation**

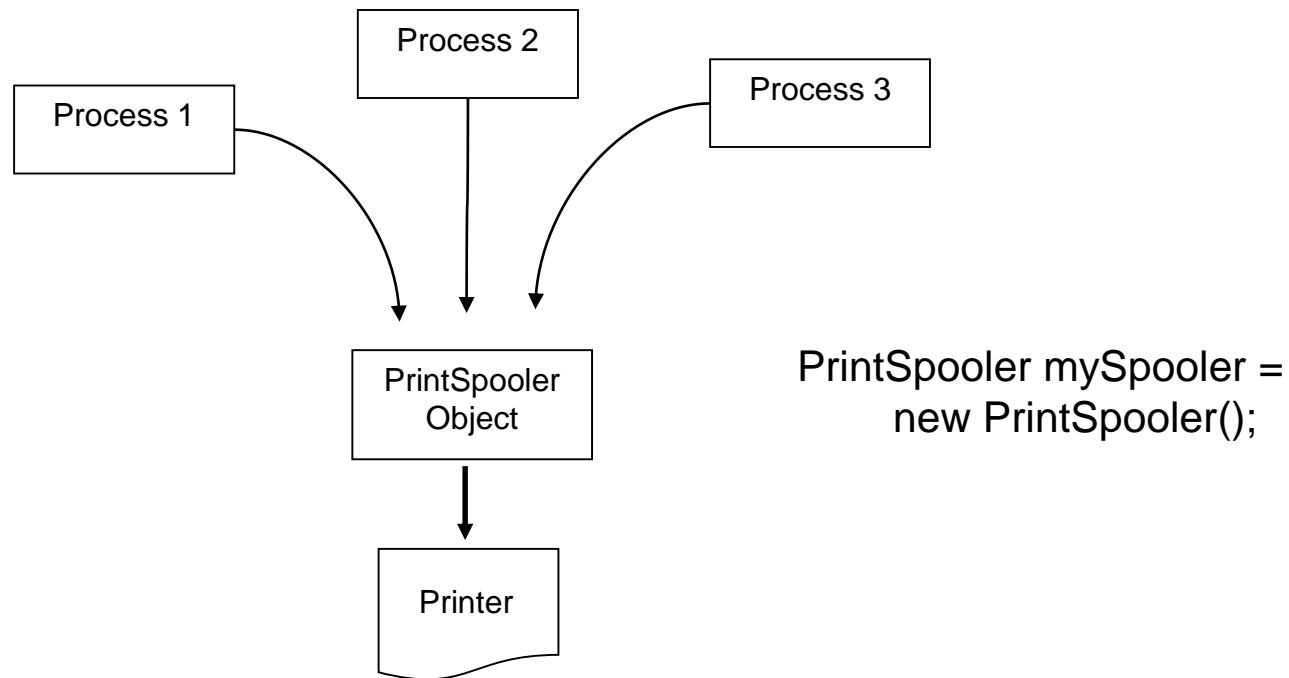
- Insure a class never has more than one instance at a time
 - Provide public access to instance creation
 - Provide public access to current instance

- **Examples**

- Print spooler
 - Audio player

PrintSpooler Example

Multiple processes should not access a single printer simultaneously



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 ...  
}
```



[Class diagram](#) exemplifying the singleton pattern.

Source:
https://en.wikipedia.org/wiki/Singleton_pattern

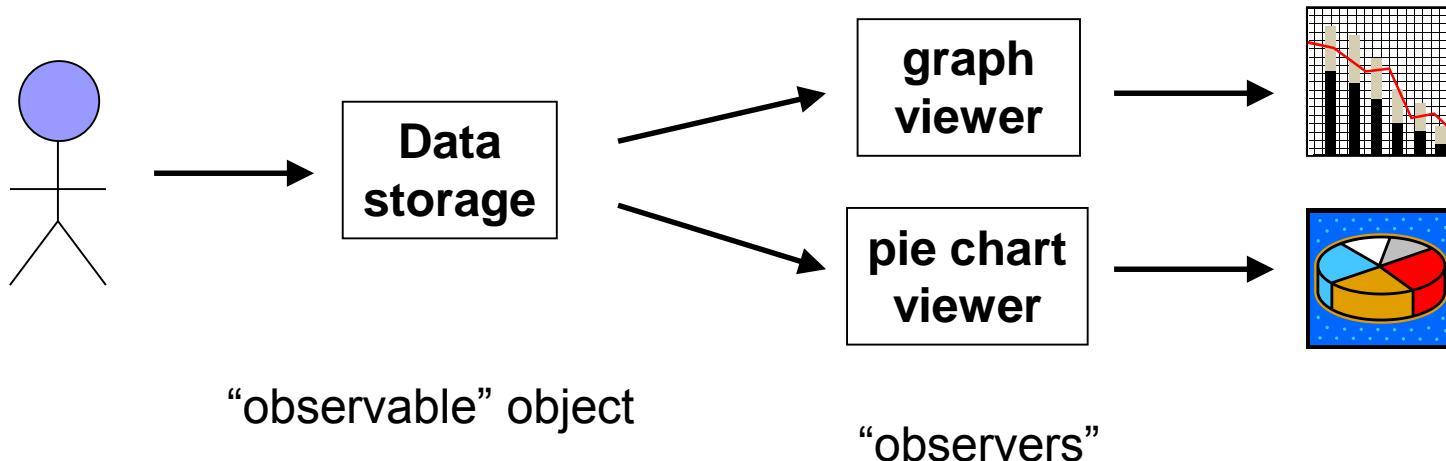
Recap: The Singleton Pattern

- This pattern involves a single class which is responsible to create an object while making sure that only single object gets created.
- This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

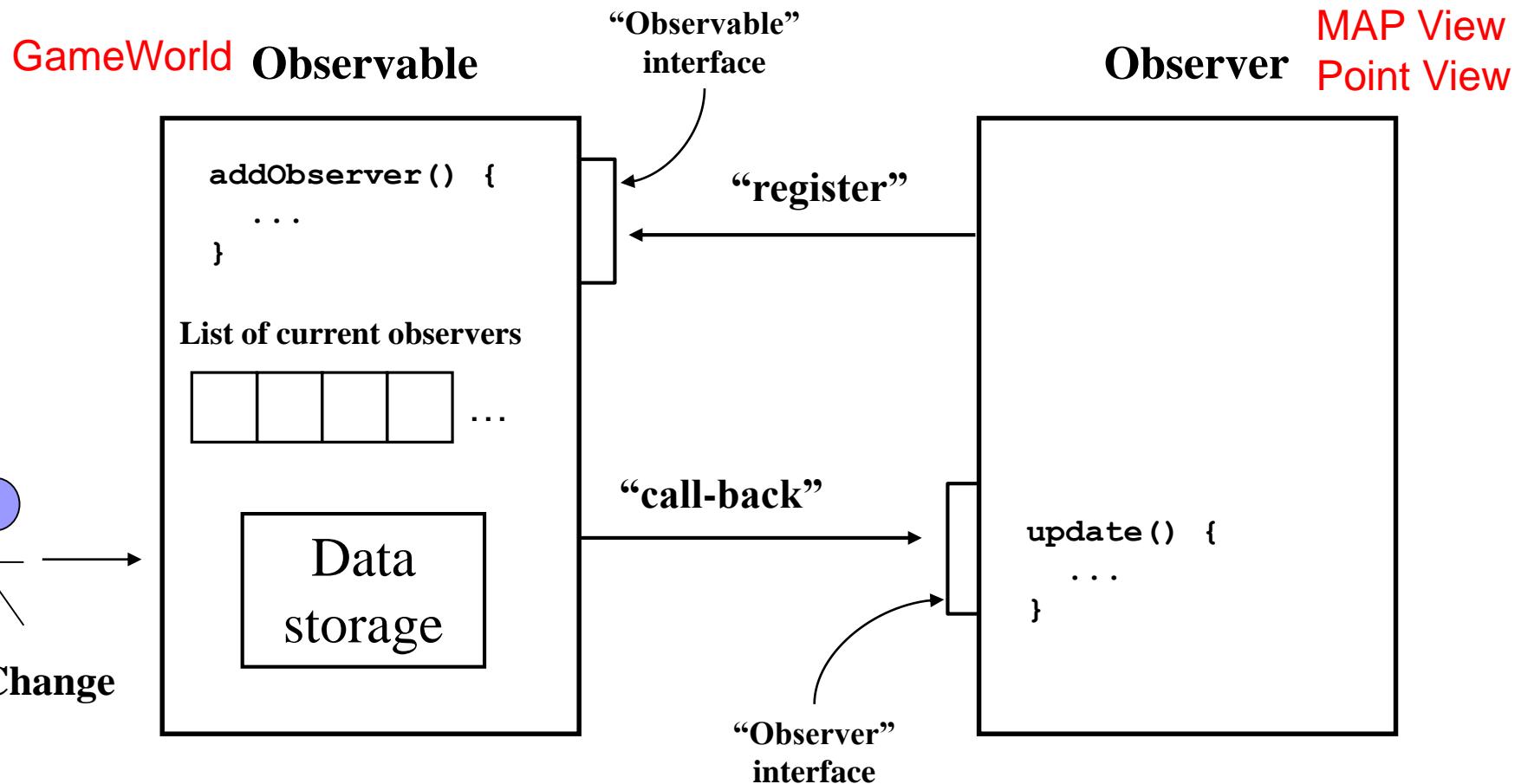
The Observer Pattern

Motivation

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



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:

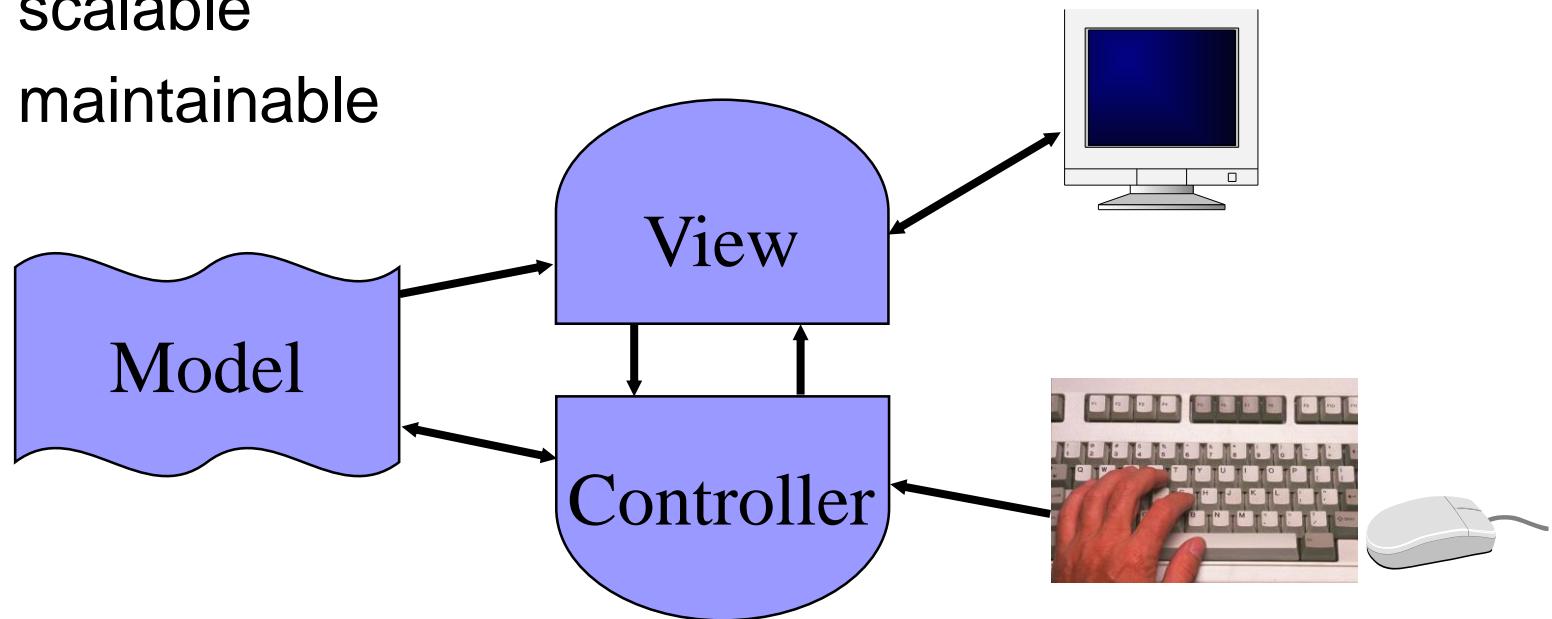
- Advantage: Provides code for **notifyObservers()** and **addObserver()**
- Disadvantage: You cannot extend from another class
- Make sure you call **setChanged()** before calling **notifyObservers()**
- **notifyObservers()** automatically calls **update()** on the list of observers that is created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

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

Model-View-Controller

Architecture

- ◆ Architecture for interactive apps
 - introduced by Smalltalk developers at PARC
- ◆ Partitions application so that it is
 - scalable
 - maintainable



Model-View-Controller

Architecture (Cont)

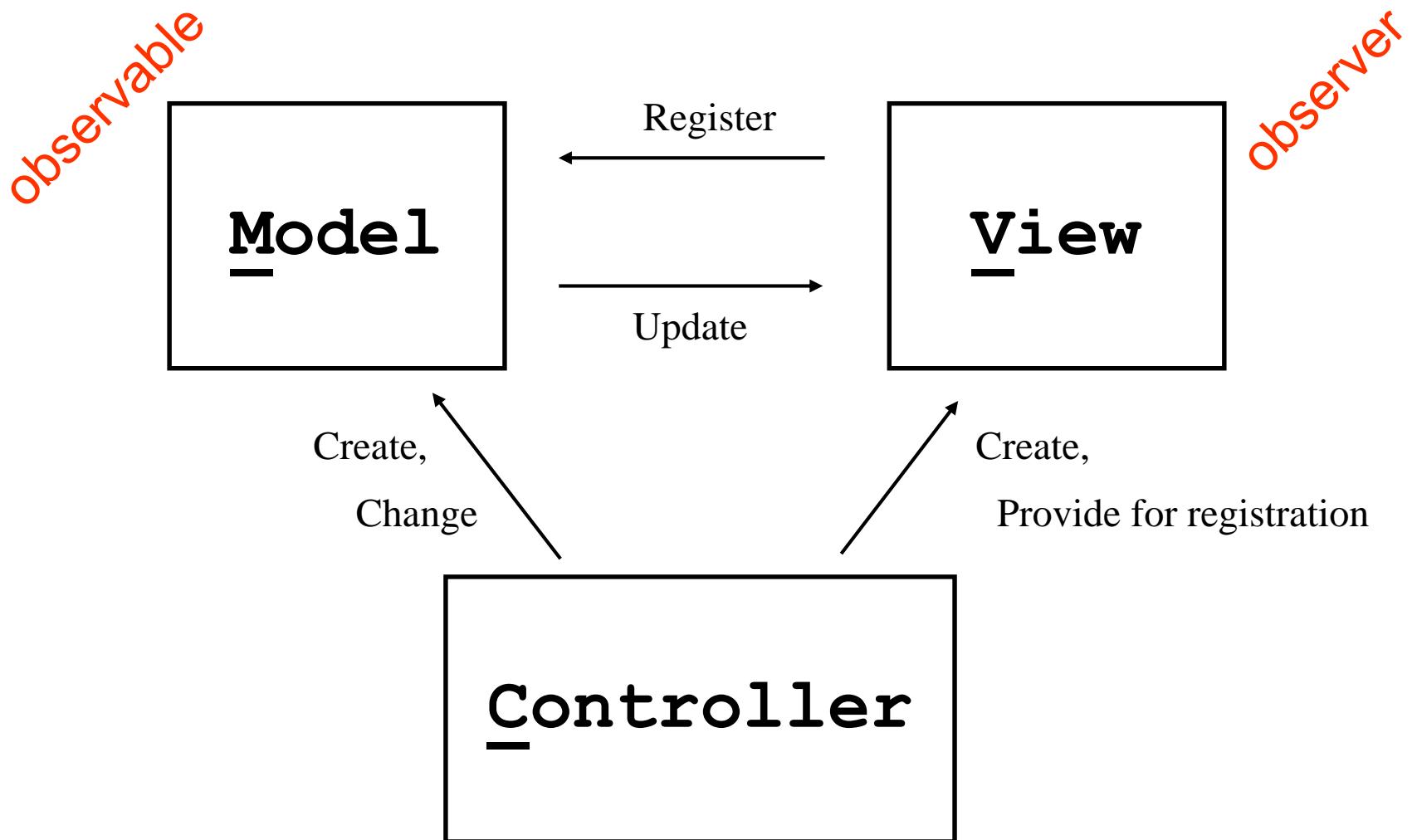
Component	Purpose	Description
Model	Maintain data	Business logic plus one or more data sources such as a database.
View	Display all or a portion of the data	The user interface that displays information about the model to the user.
Controller	Handle events that affect the model or view	The flow-control mechanism means by which the user interacts with the application.

Model-View-Controller

design pattern (Cont)

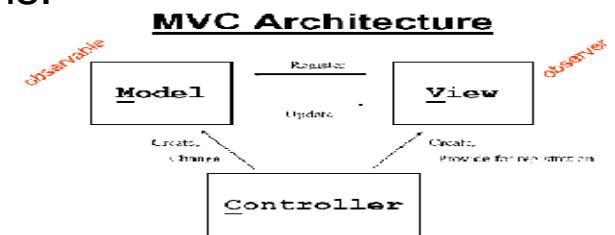
Component	In our assignment # 1 context (Spring 2019)
Model (GameWorld)	A game in turn contains several components, including (1) a GameWorld which holds a collection of game objects and other state variables . Later, we will learn that a component such as GameWorld that holds the program's data is often called a model.
View (Future: Map and Score Views)	In this first version of the program the top-level Game class will also be responsible for displaying information about the state of the game. In future assignments we will learn about a separate kind of component called a view which will assume that responsibility.
Controller (Game)	The top-level Game class also manages the flow of control in the game (such a class is therefore sometimes called a controller). The controller enforces rules such as what actions a player may take and what happens as a result. This class accepts input in the form of keyboard commands from the human player and invokes appropriate methods in the game world to perform the requested commands – that is, to manipulate data in the game model.

MVC Architecture



Design Pattern vs Architectural Pattern

- **Design patterns** are usually associated with **code level commonalities (Classes)**. It provides various schemes for refining and building smaller subsystems.
 - Design pattern answers reoccurring problems in software construction.
- **Architectural patterns** are seen as commonality at higher level than design patterns. Architectural patterns are high-level strategies that concerns **large-scale components**, the global properties and mechanisms of a system.
 - Architectural pattern deals with how the major parts of the system fit together, how messages and data flow through the system, and other structural concerns.



```
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  
    }  
}
```

Skeleton Code

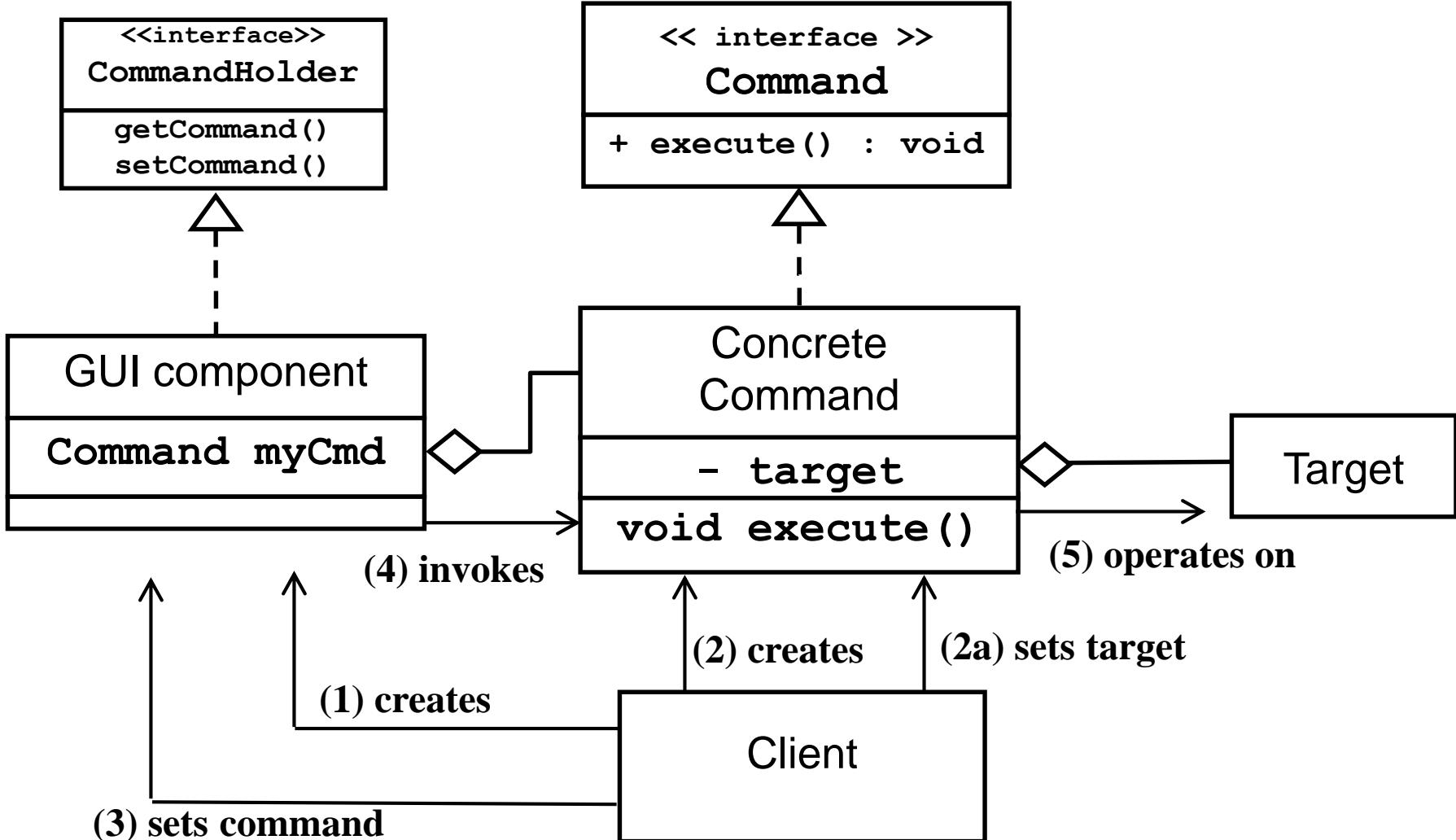
Part II – Design Pattern

The Command Pattern

Motivation

- o Need to avoid having multiple copies of the code that performs the same operation invoked from different sources
- o Desire to separate code implementing a command from the object which invokes it
- o 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 subclass 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!

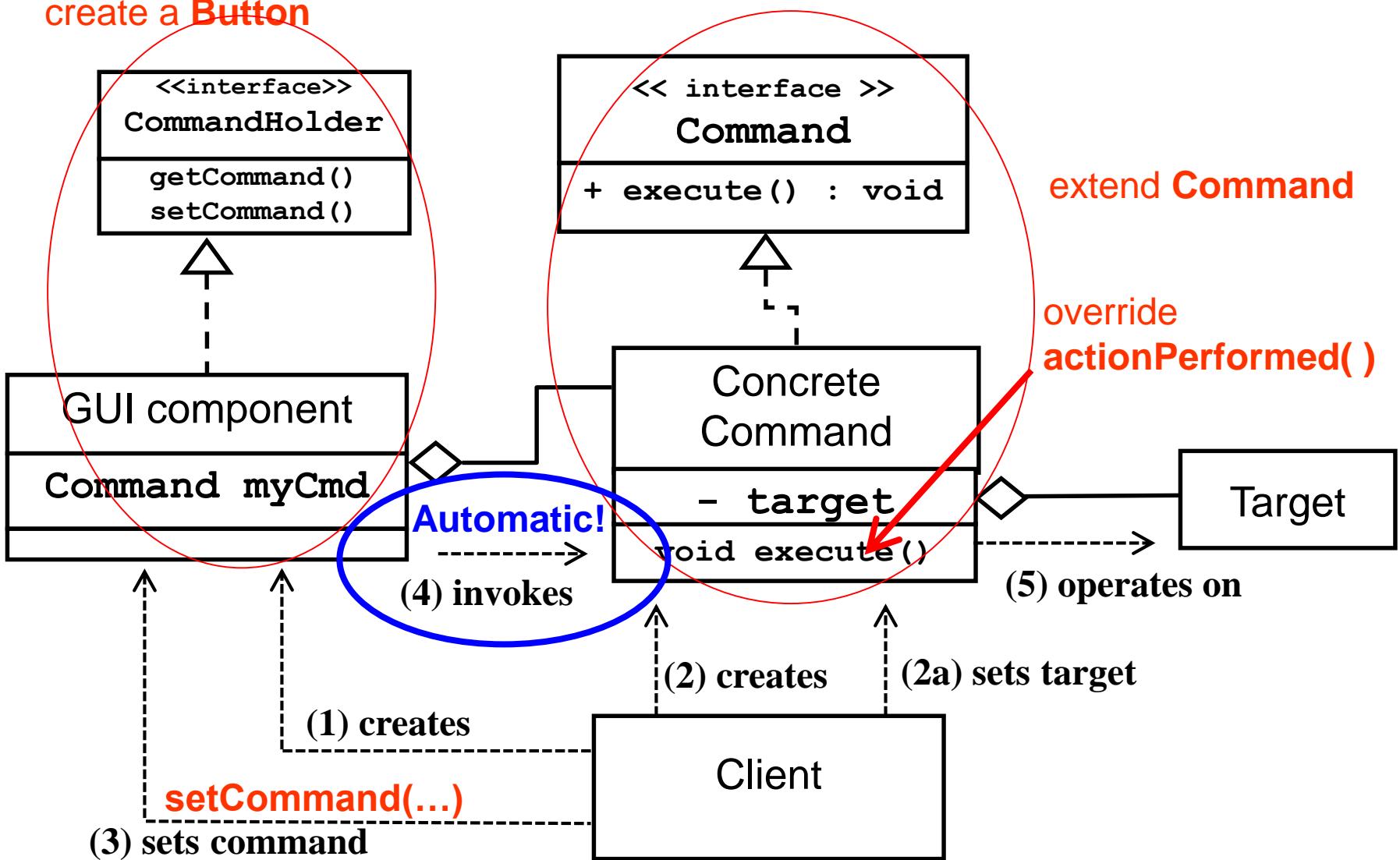
Adding Commands to Title Bar

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
- `addCommandToXXX` i.e. `addCommandToRightBar`

Command Pattern – CN1

create a Button



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 buttons 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 command 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

- o A variety of algorithms exists to perform a particular operation
- o 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?

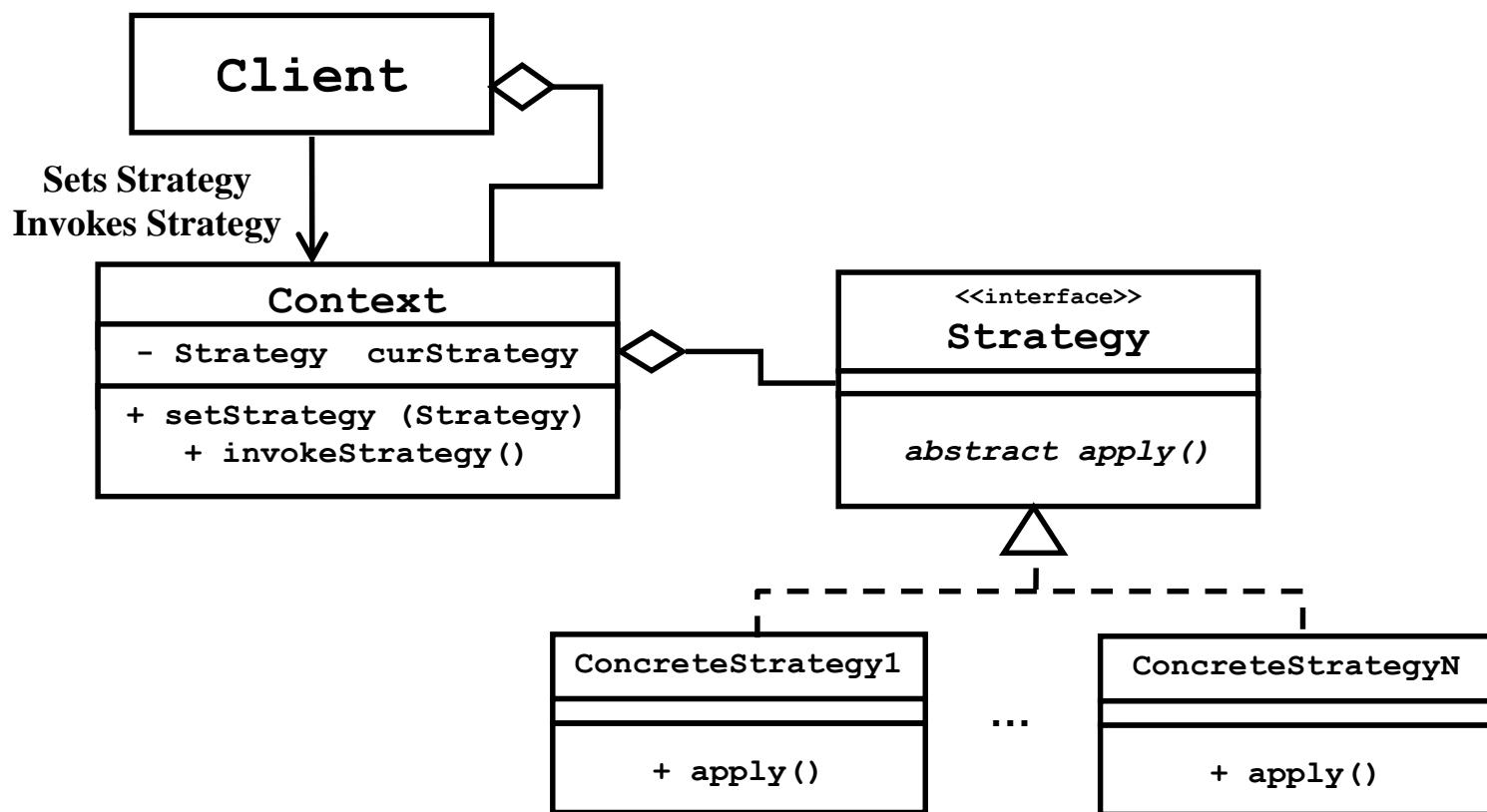
Changing or adding a plan requires changing the client!

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

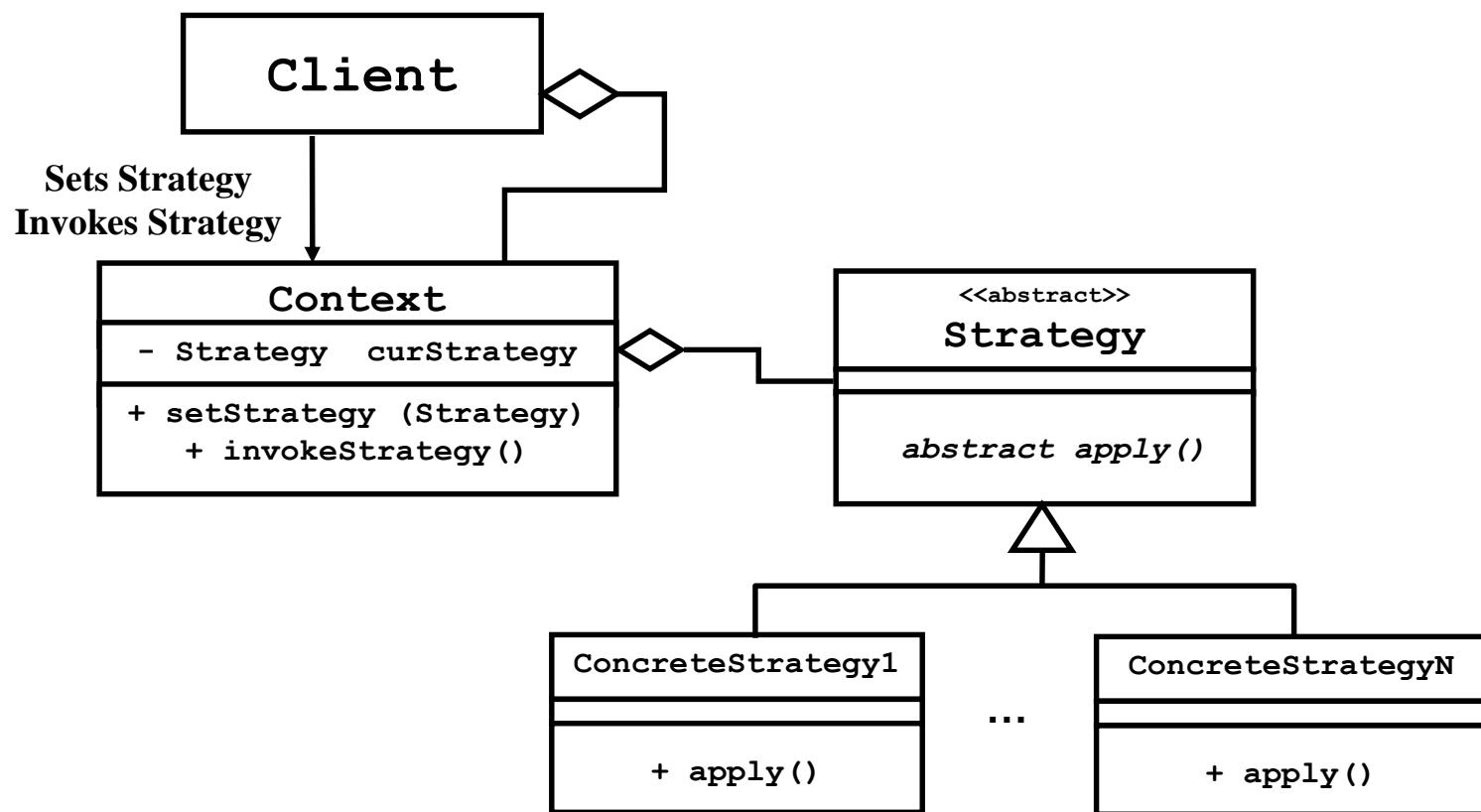
Strategy Pattern Organization

- 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)
```

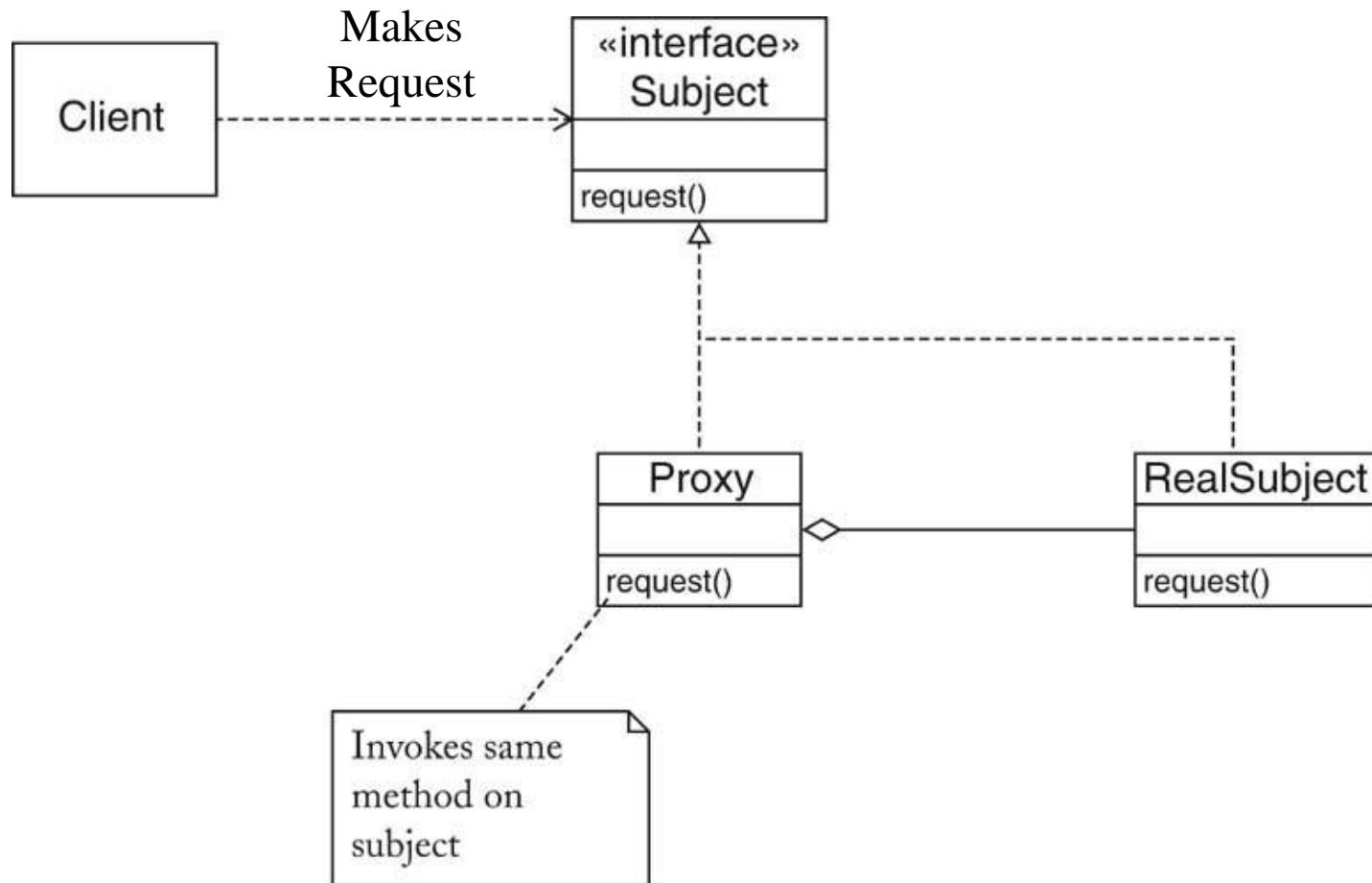
The Proxy Pattern

- 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());

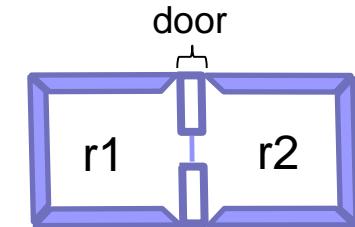
        r2.setSide(NORTH, new Wall());
        r2.setSide(EAST, new Wall());
        r2.setSide(SOUTH, new Wall());
        r2.setSide(WEST, theDoor);

        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()...)
}

```



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) {

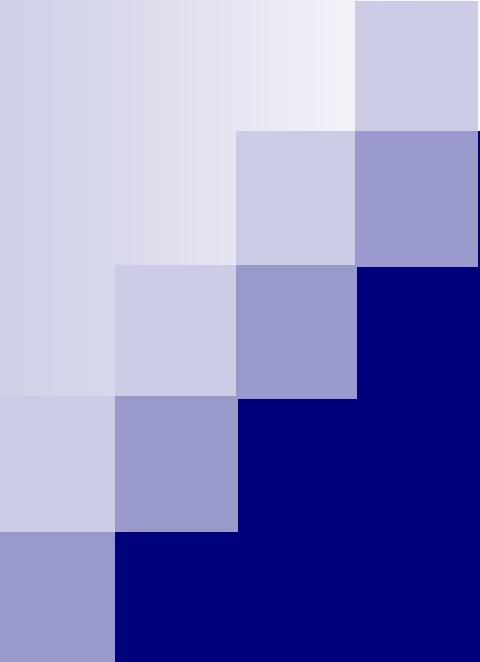
        //create the spell necessary to enter the enchanted room
        Spell spell = makeSpell() ;

        //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

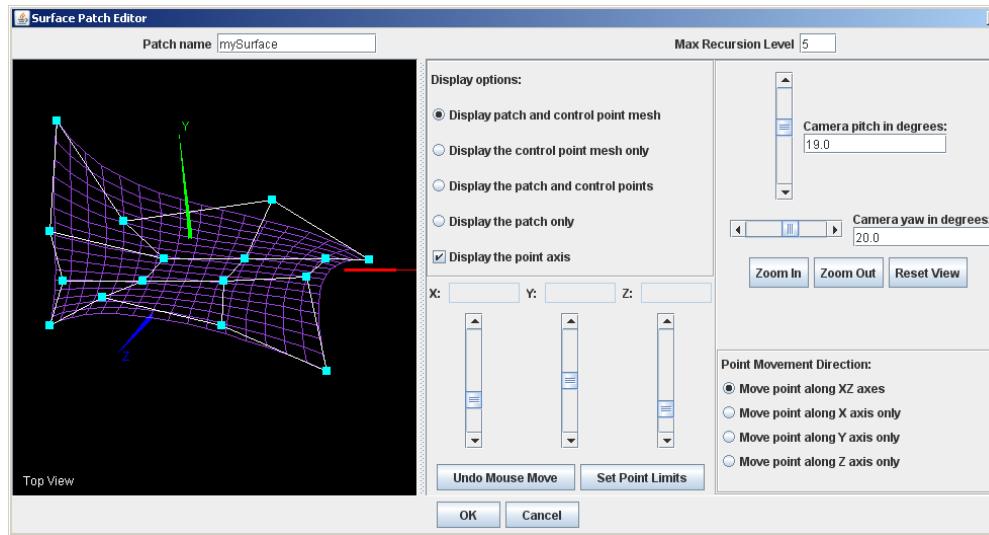
- **Displays**
- **The UI Package of CN1**
- **UI Components: Form, Button, Label, Checkbox, ComboBox, TextField ...**
- **Layout Managers**
- **Containers**
- **Side Menus**

Resources:

- (1) <https://www.codenameone.com/manual/basics.html>
- (2) Sample UI Code in Canvas (Week 5)

Modern Program Characteristics

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



GUI Frameworks

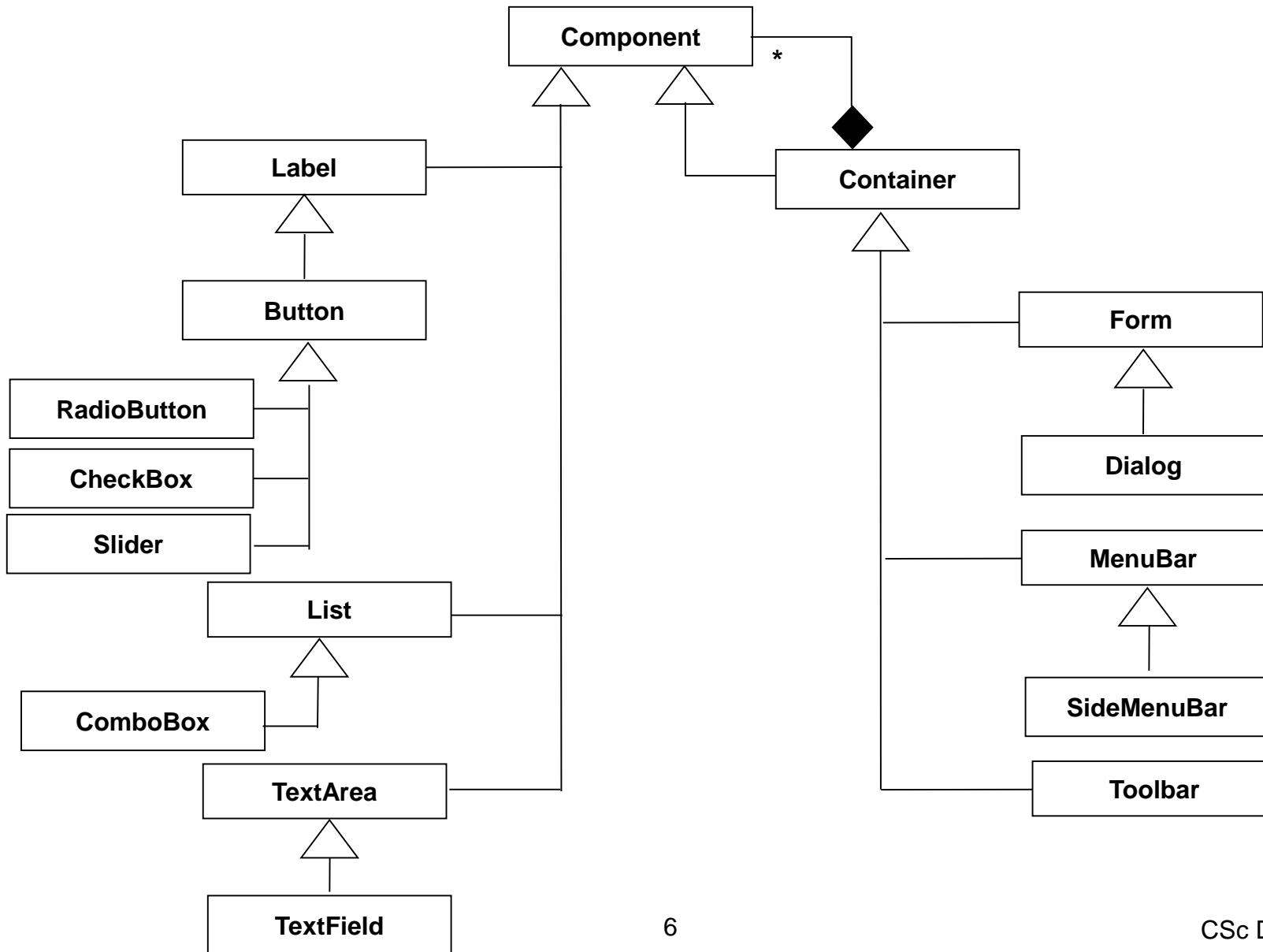
- Collection of classes that take care of low-level details of drawing “things” on screen. Provides:
 - A set of reusable screen components
 - “Component”: an object having a graphical representation
 - Usually has the ability to interact with the user
 - An event mechanism connecting “actions” to “code”
 - 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)

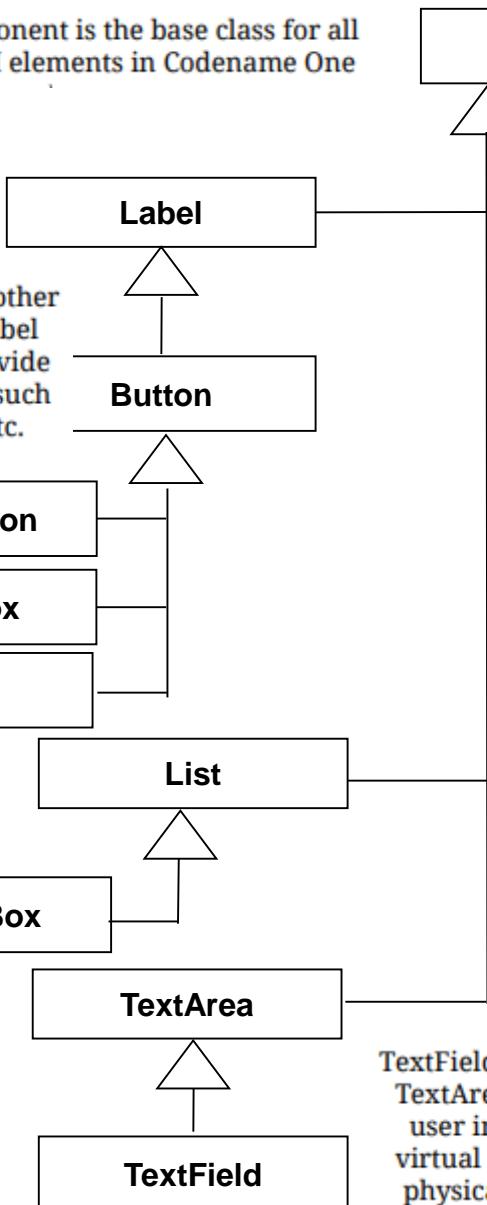
Every button, label or element you see on the screen in a Codename One application is a Component!

Important CN1- UI Components

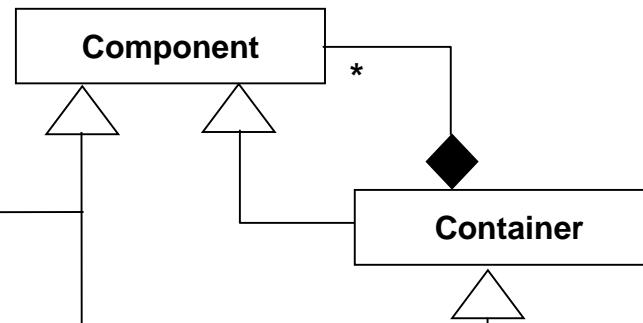


Important CN1- UI Components

Component is the base class for all the UI elements in Codename One

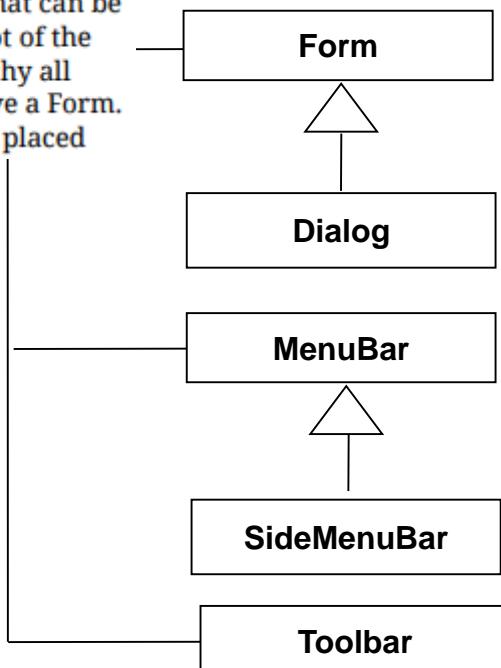


Button and quite a few other classes derive from Label this allows them to provide common functionality such as icons, alignment etc.



Container is a Component that can hold within it other components. Since a Container is a Component itself it can hold other Containers within. This allows elaborate hierarchies

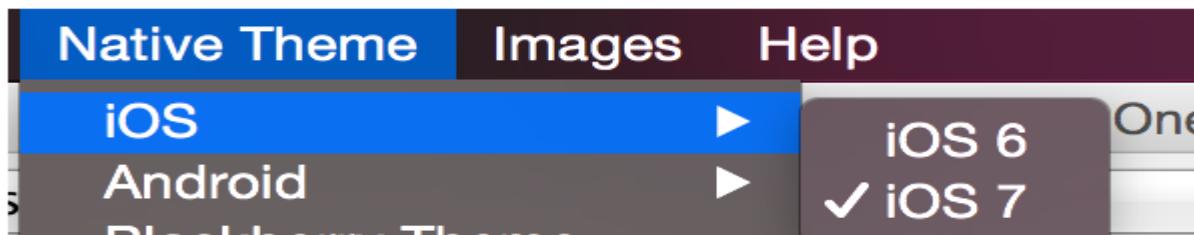
Form is a Container that can be “shown” it's the root of the Container hierarchy all applications must have a Form. It's where the UI is placed



TextField derives from TextArea both allow user input using a virtual keyboard (or physical keyboard)

CN1 Theme, Component, Style

Theme: Codename One themes are pluggable CSS like elements that allow developers to determine/switch the look of the application in runtime.



Component: Every visual element within Codename One is a Component, the button on the screen is a Component and so is the Form in which it's placed. This is all represented within the Component, which is probably the most central class in Codename One.

Style: Every component has 4 standard style objects associated with it: Selected, Unselected, Disabled & Pressed.

Only one style is applicable at any given time and it can be queried via the `getStyle()` method. A style contains the colors, fonts, border and spacing information relating to how the component is presented to the user

An Example of a form in CN1

- Content Pane - this is literally the body of the **Form**. When we **add** a **Component** into the **Form** it goes into the content pane. Notice that Content Pane is scrollable by default on the Y axis!
- Title Area - we can't add directly into this area. The title area is managed by the **Toolbar** class. **Toolbar** is a special component that resides in the top portion of the form and abstracts the title design. The title area is broken down into two parts:
 - Title of the **Form** and its commands (the buttons on the right/left of the title)
 - Status Bar - on iOS the area on the top includes a special space so the notch, battery, clock etc. can fit. Without this the battery indicator/clock or notch would be on top of the title

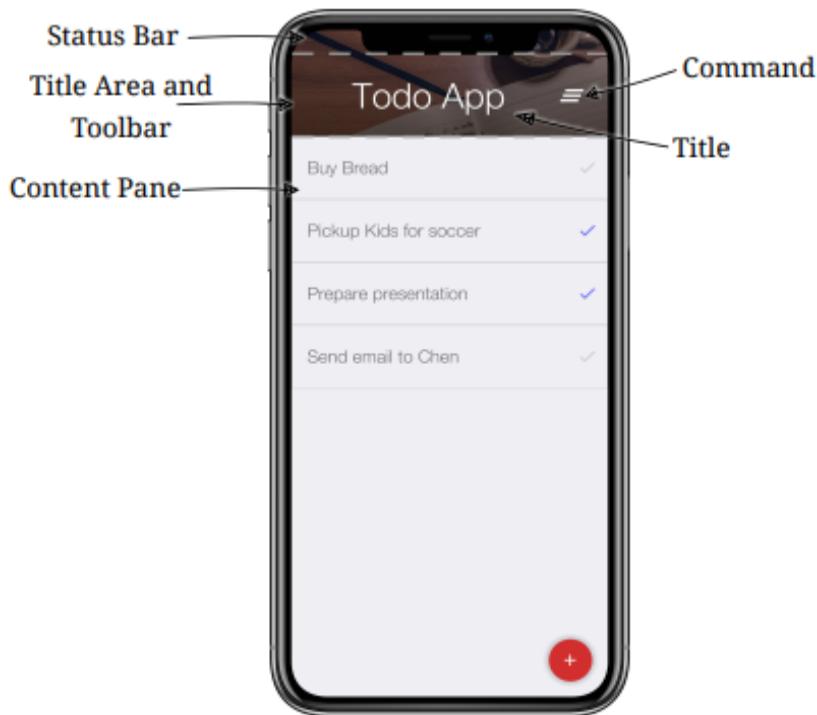
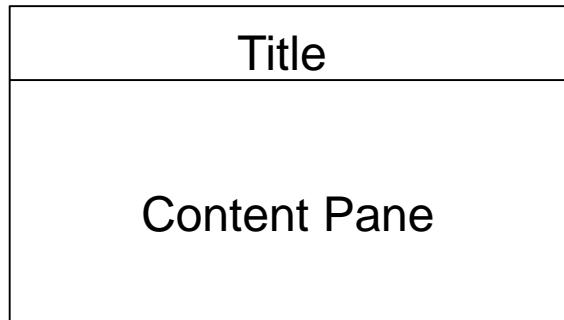


Figure 2. 4. Structure of a Form

Credits: Create an Uber Clone in 7 Days (Shai Almog – Codename One Academy)

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(manual)" 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();  
    }  
}
```

Note: Show a starter demo in class next.

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:
            // This helps to quit the application
            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.

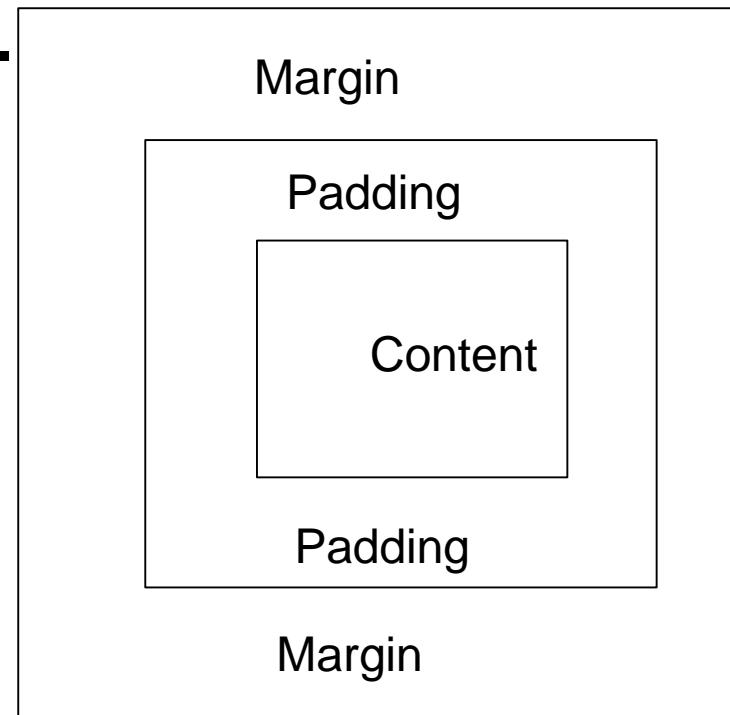
Adding Components to Form

```
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.png");  
  
        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.png");  
  
        // [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:

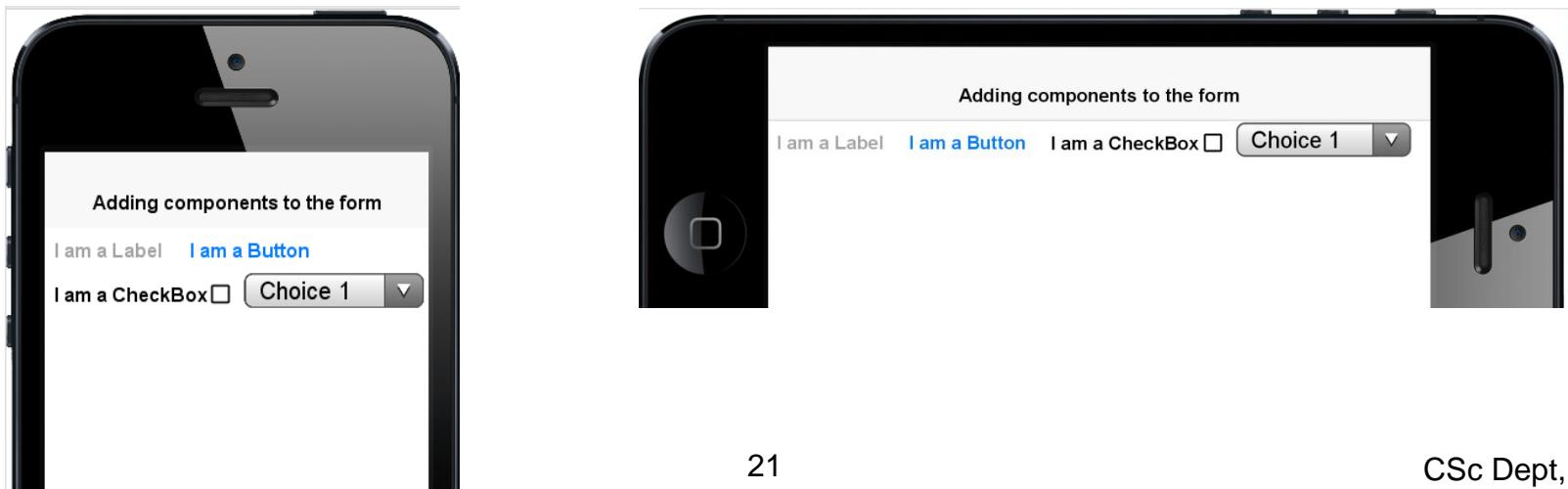
```
myContainer.setLayout( new BorderLayout() );
```
- Components can have a preferred size
 - `setPreferredSize()` of `Component` is deprecated
 - override `calcPreferredSize()` of `Component` to reach similar functionality (do not use this in the assignments)
 - Layout managers *may or may not* respect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

Layout Managers (cont.)

- Example: **FlowLayout**

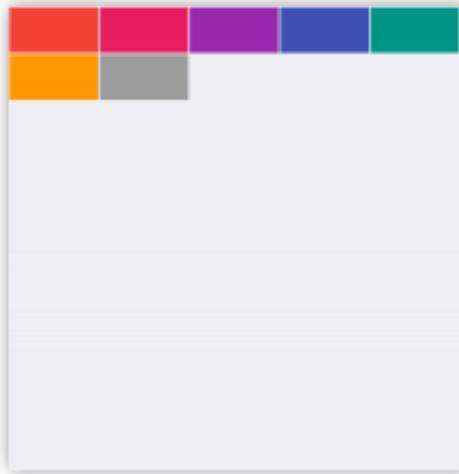
- Arranges components left-to-right, top-to-bottom (by default)
- Components appear in the order they are added
- Respects *preferred size*
- Components that don't fit may be *hidden*
- You can center components in the component by using:

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



Layout Managers (cont.)

Flow Layout sizes components based on their preferred size and arranges them from left to right. When we reach the end of the line it breaks a line



Flow layout has several modes including a center mode that center aligns elements. It can align elements to the right and align vertically as well



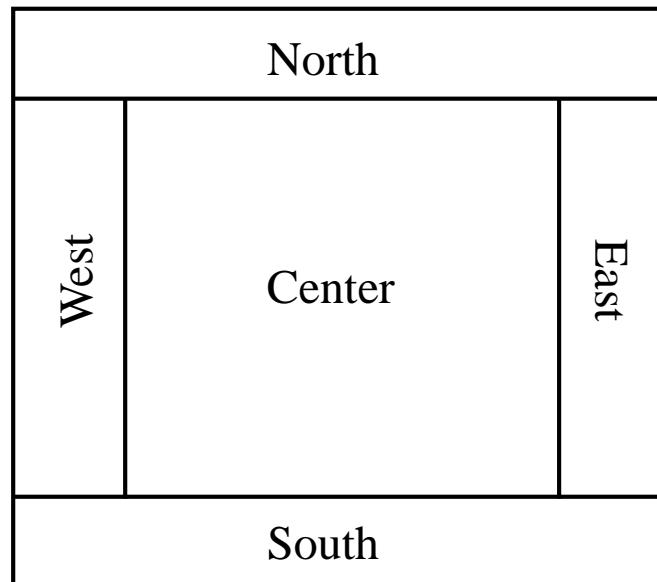
Credits: Create an Uber Clone in 7 Days (Shai Almog – Codename One Academy)

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);
```



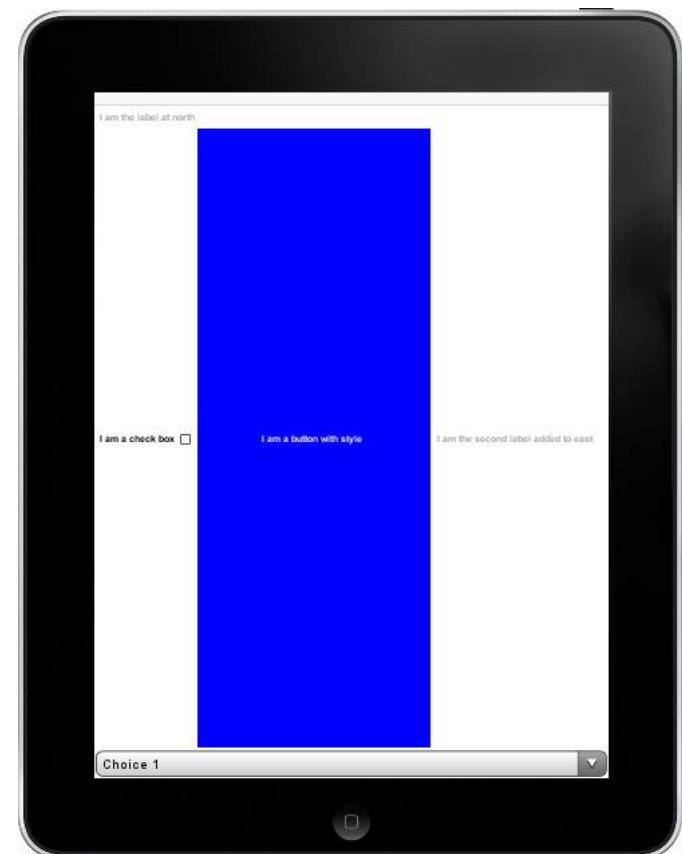
Layout Managers (cont.)

- **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.

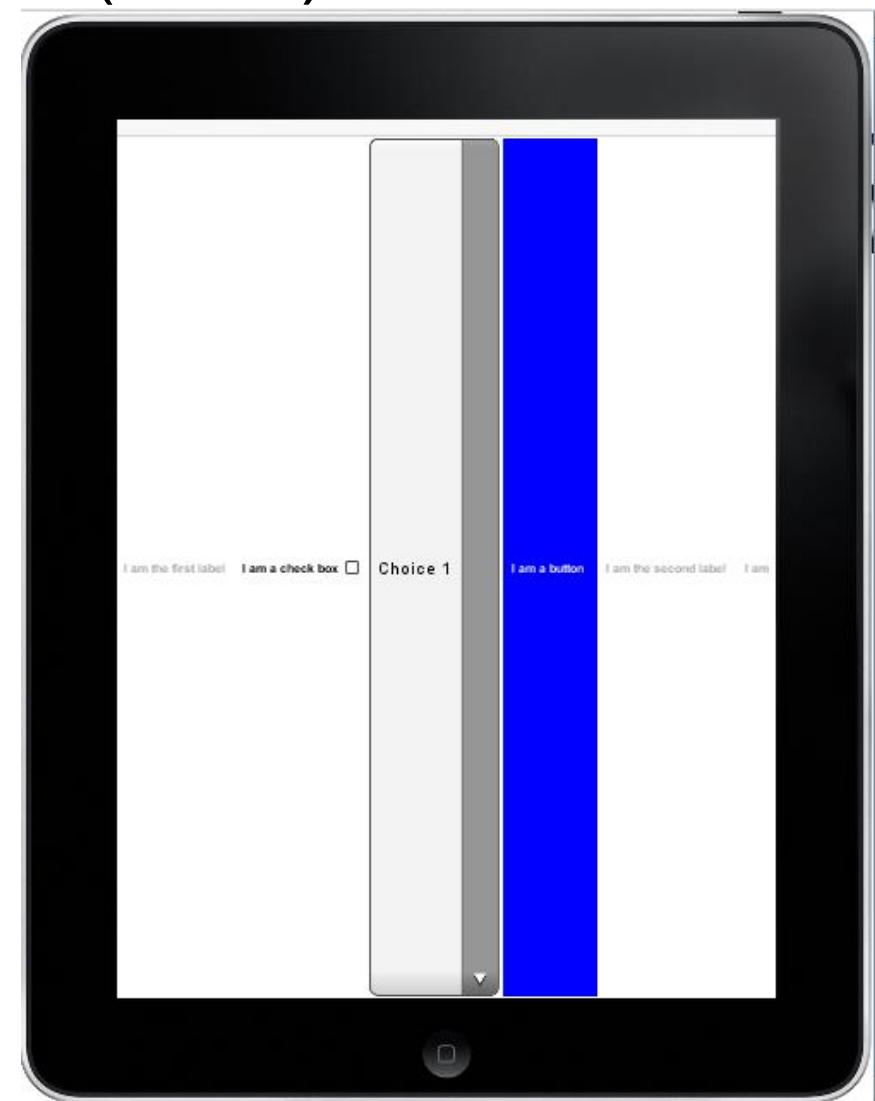
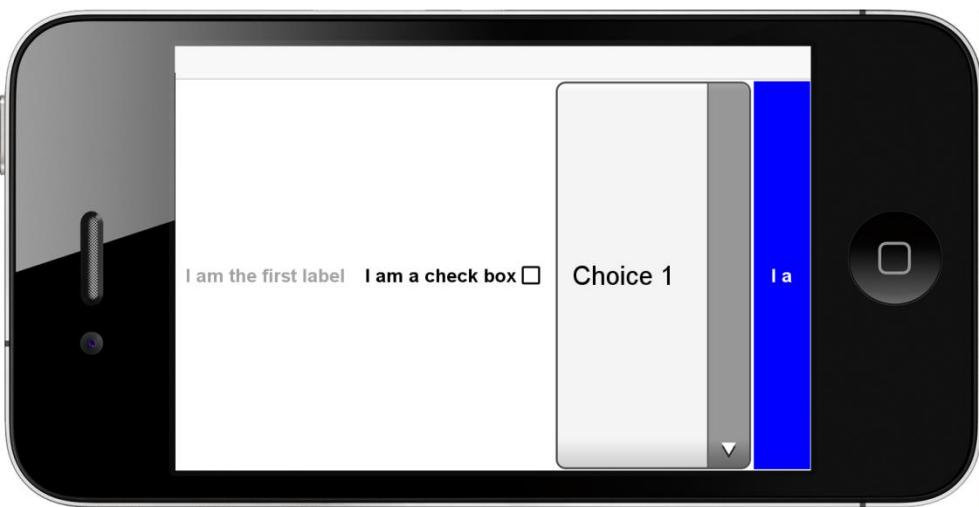
Layout Managers (cont.)

- 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

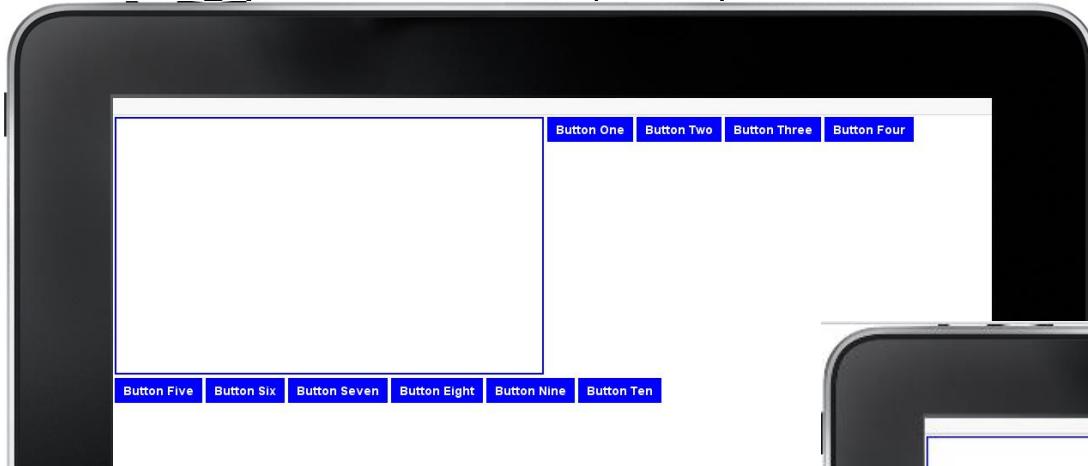
----- below is the code for a form with box layout

```
//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

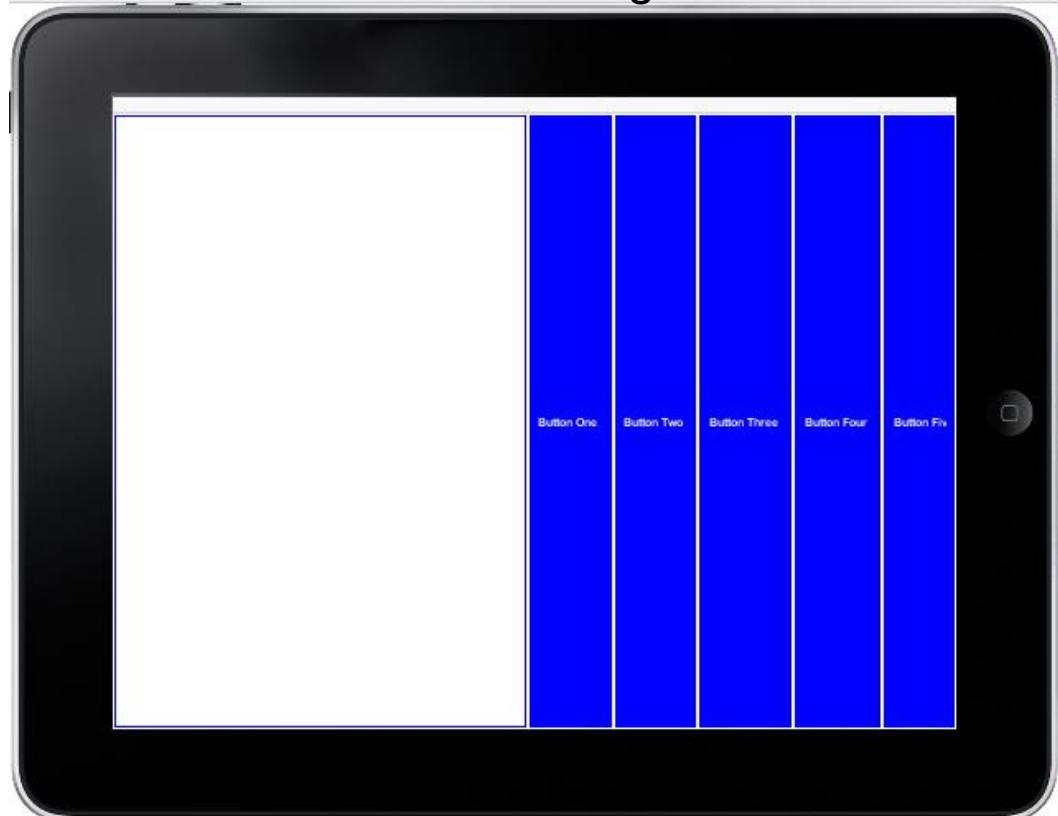
Layout Managers (cont.)

Setting preferred size (cont.):



BoxLayout (below)
Respects preferred width
but not height...

FlowLayout (above) respects both
preferred width and 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

Overview

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

Source: <https://www.codenameone.com/manual/basics.html>

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**)

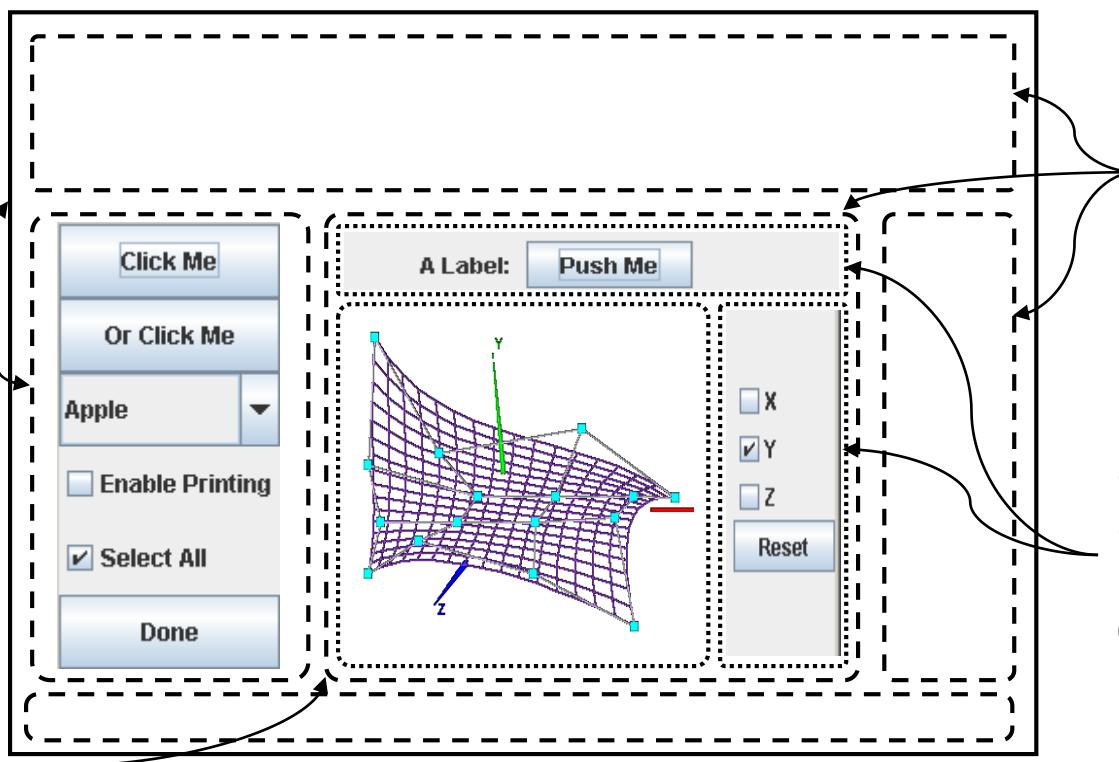
**Form with
BorderLayout
(Form is-a
Container)**

**West Container
with components
in BoxLayout**

**Center
Container with
BorderLayout**

**Containers
(dashed) in
N/S/E/W/C of
Form**

**Containers
(dotted) in
Center
Container**



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 (1)"));
leftContainer.add(new Button("Click Me (1)"));
leftContainer.add(new ComboBox("Choice 1","Choice 2","Choice 3"));
leftContainer.add(new CheckBox("Enable Printing (1)"));
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));

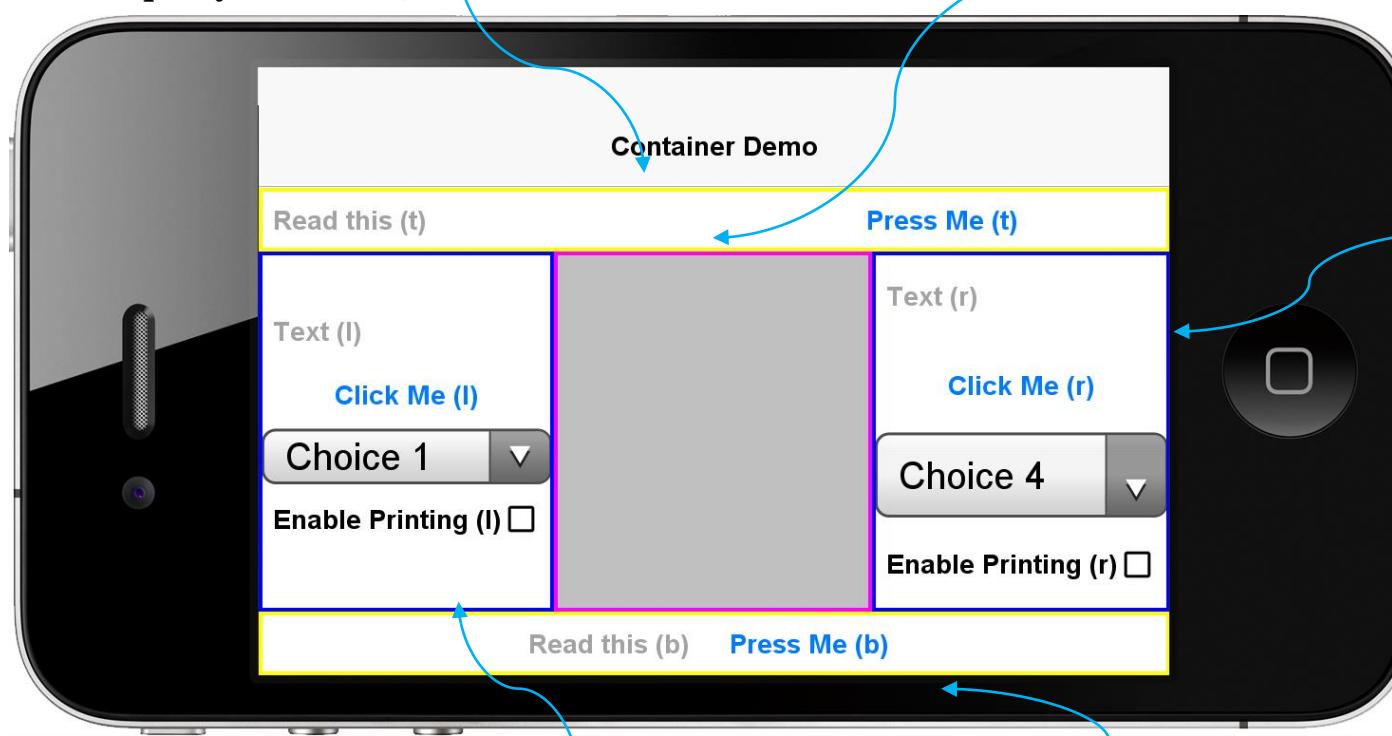
add(BorderLayout.CENTER,centerContainer);

//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)

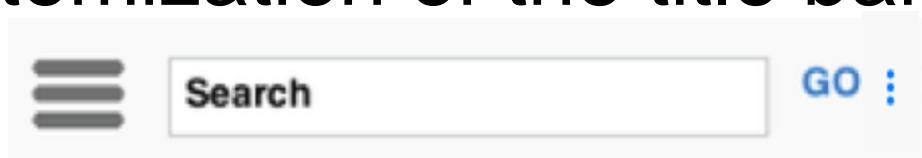
Empty Container in Center with
(with light gray background)



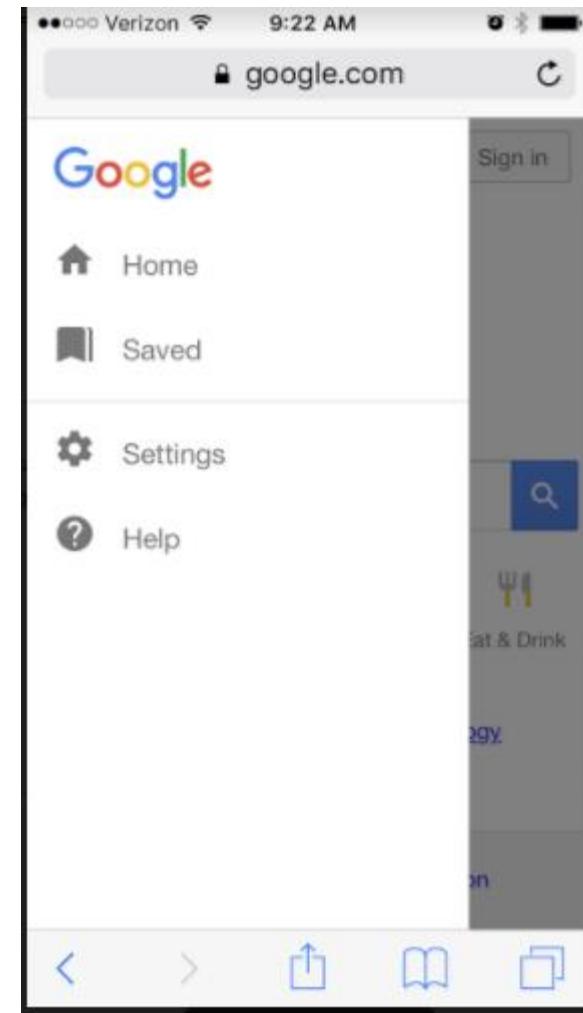
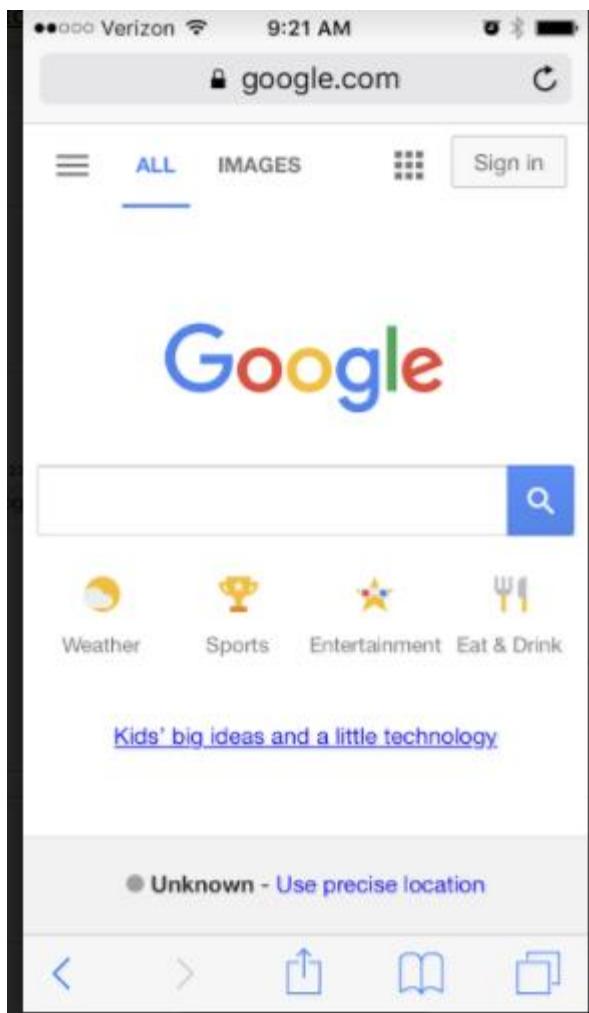
Container (with padding) in
West with BoxLayout (components
are positioned 50 pixels below the top
border of the container)

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 form 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)

Example Toolbar



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 deprecated

//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");

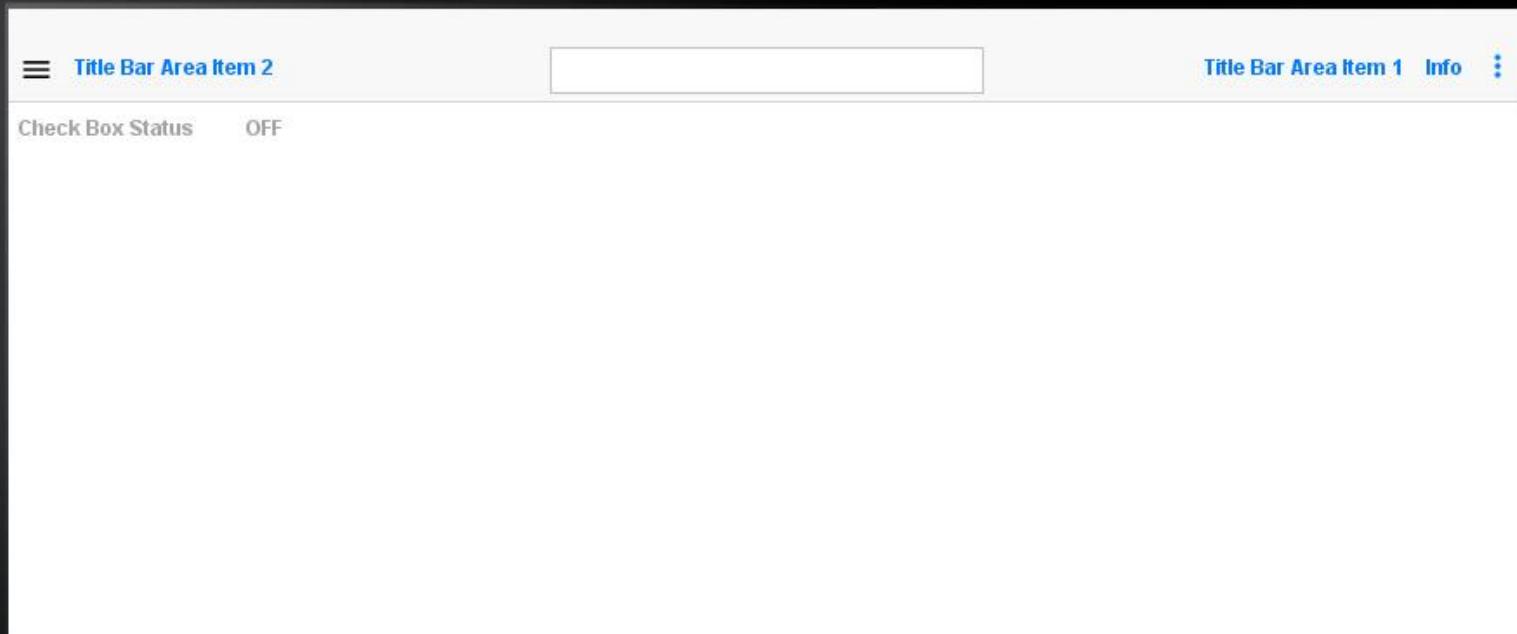
myToolbar.addCommandToRightBar(titleBarAreaItem1);

//add an "empty" item to left side of title bar area

Command titleBarAreaItem2 = new Command("Title Bar Area Item 2");

myToolbar.addComponentToSideMenu(checkSideMenuComp); //... [add other side menu,
overflow menu, and/or title bar area items]
```

Adding Items to Title Bar (cont.)



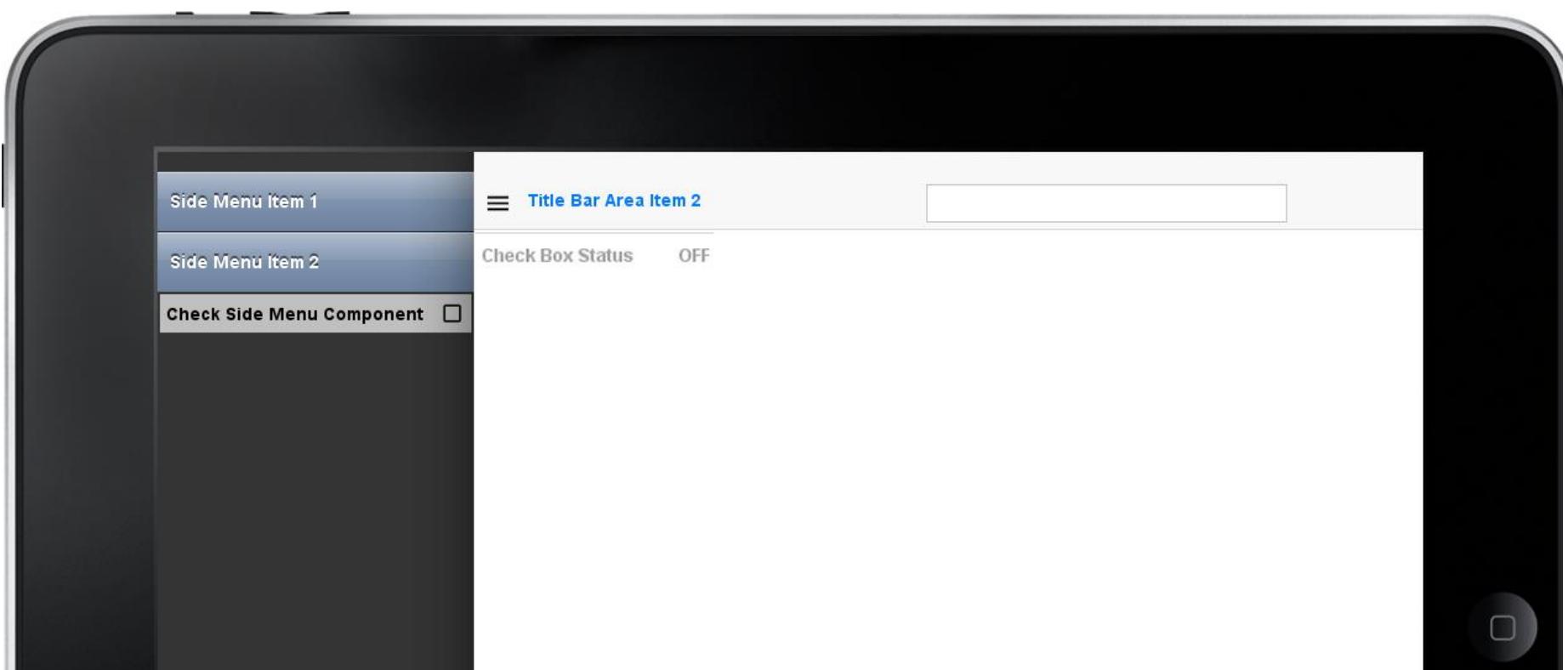
Complex Menus

- 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

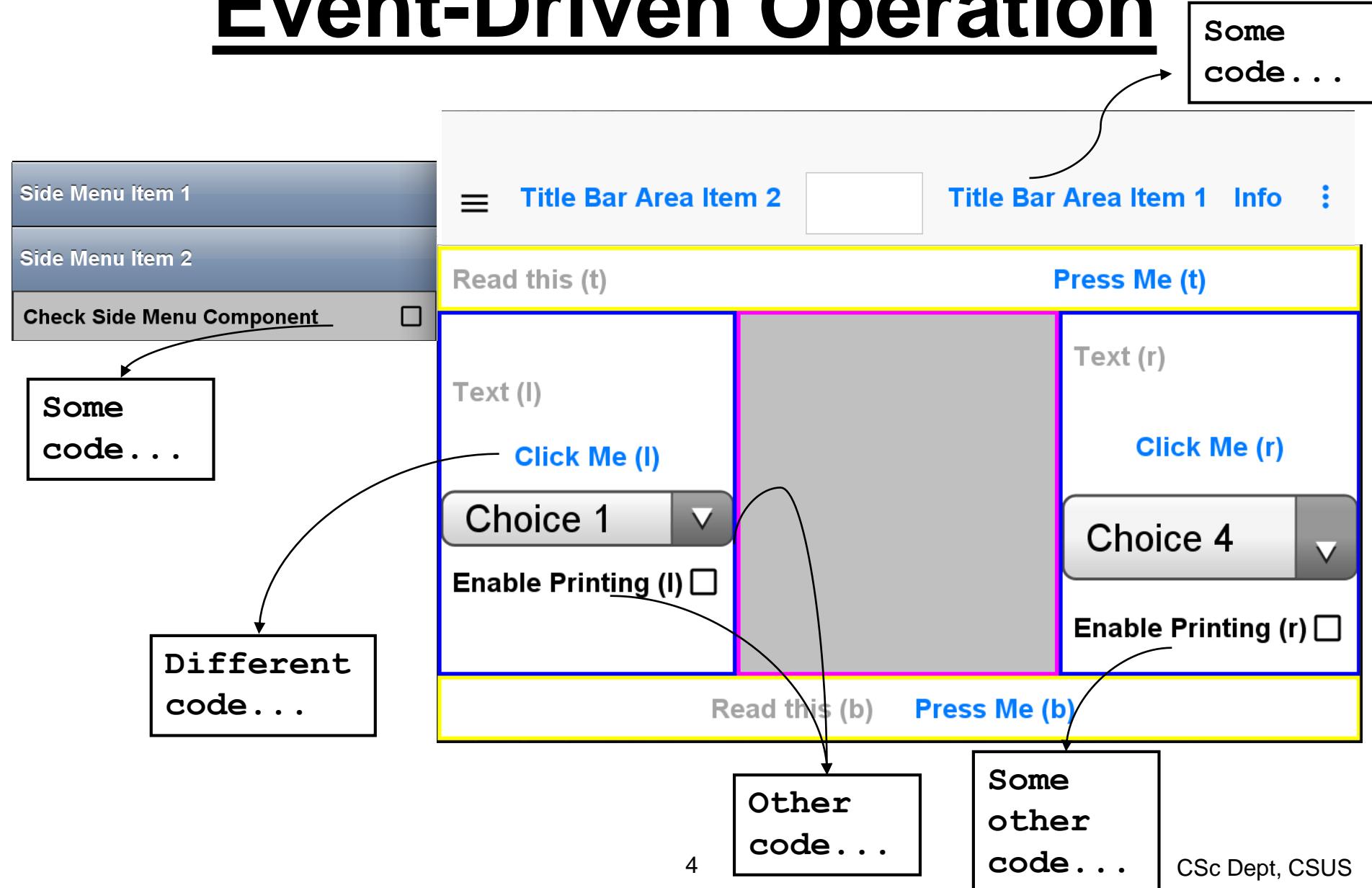
- Traditional program organization:

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

- Event-driven program organization:

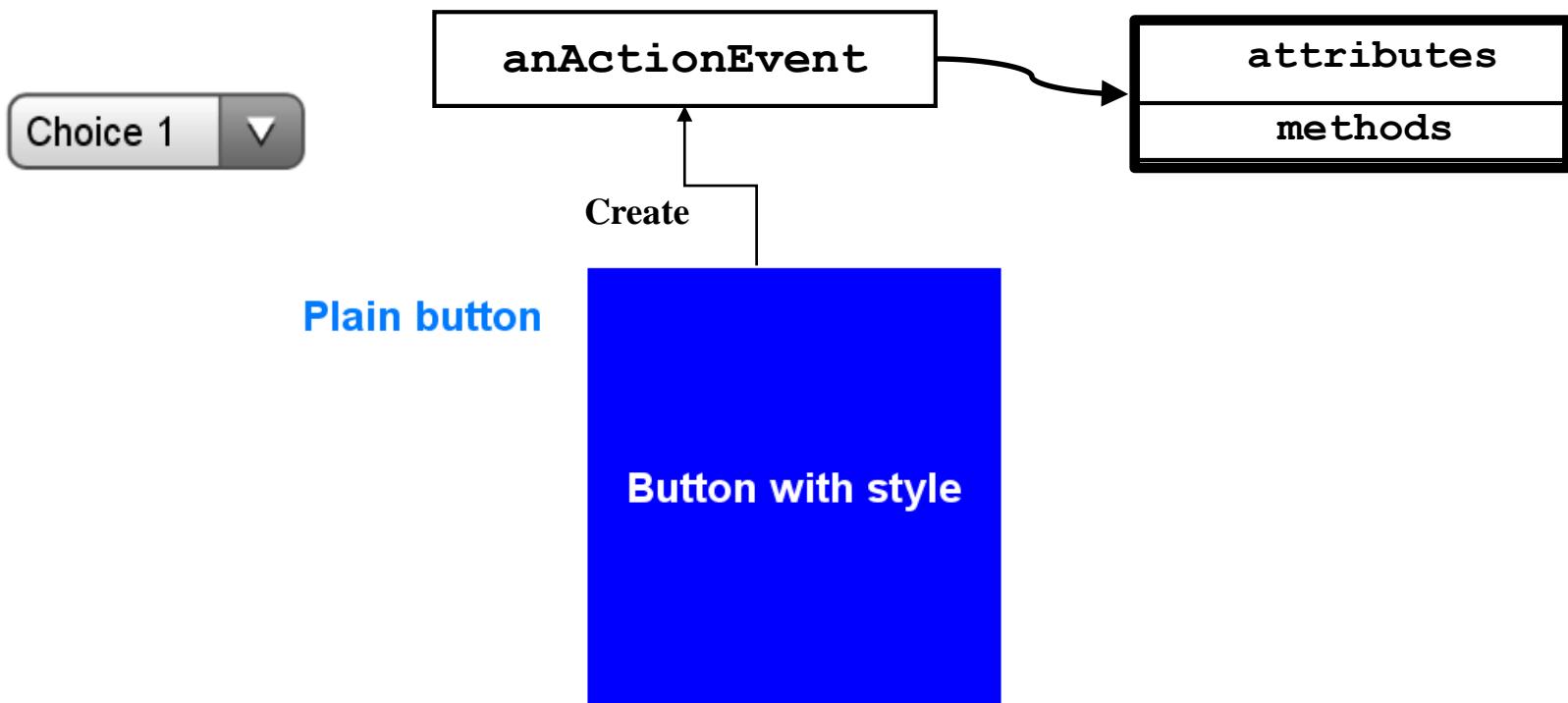
```
create a form ;
create some controls (buttons, etc.) ;
add controls to form ;
make the form visible ;
```

Event-Driven Operation



Event Objects

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



Check Side Menu Component

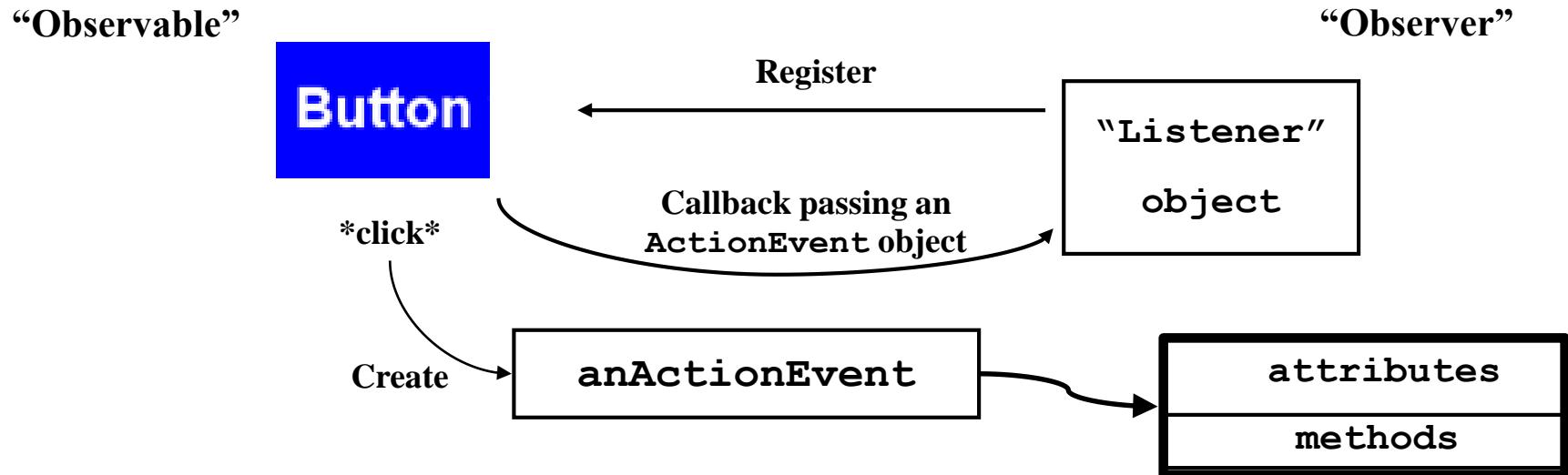


Event Objects (cont.)

- 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...");
    }

}
```

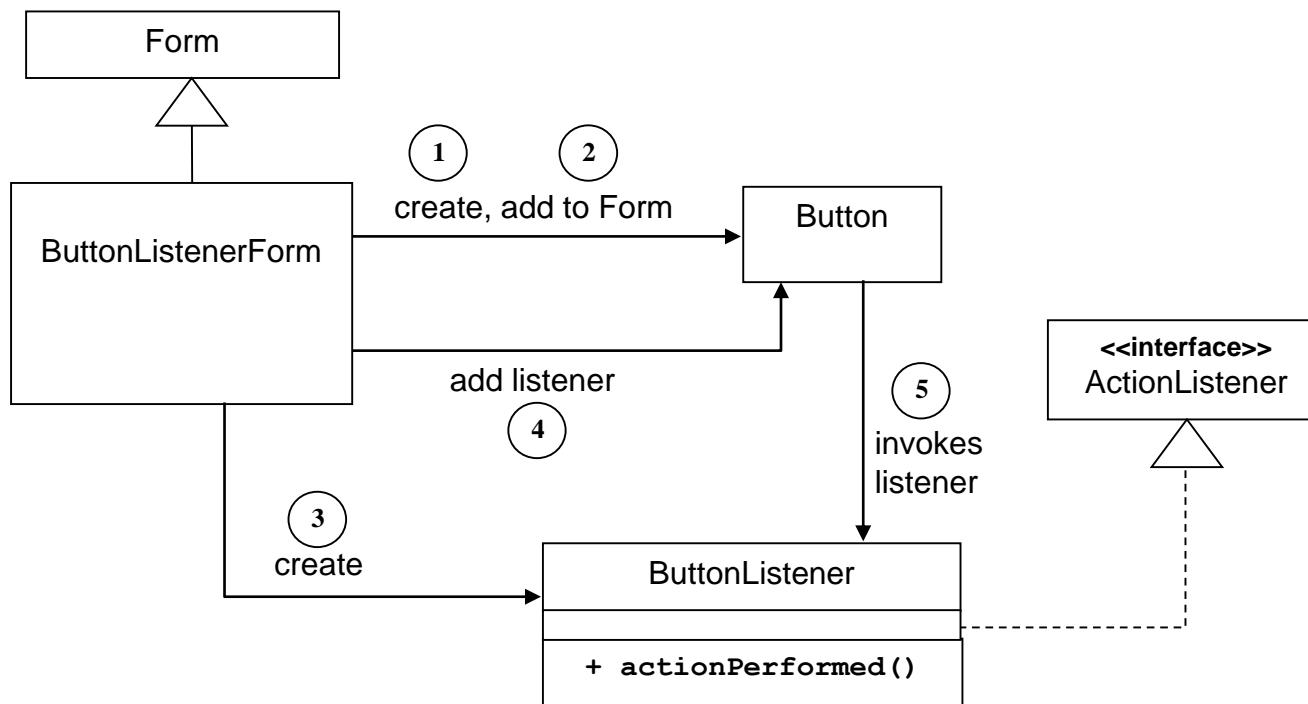
Using the Listener

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!

Form

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

```
EventHandler code:  
{  
    ...  
}
```

Create

addListener(*this*)

click event

Callback

ActionListener Frame 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);

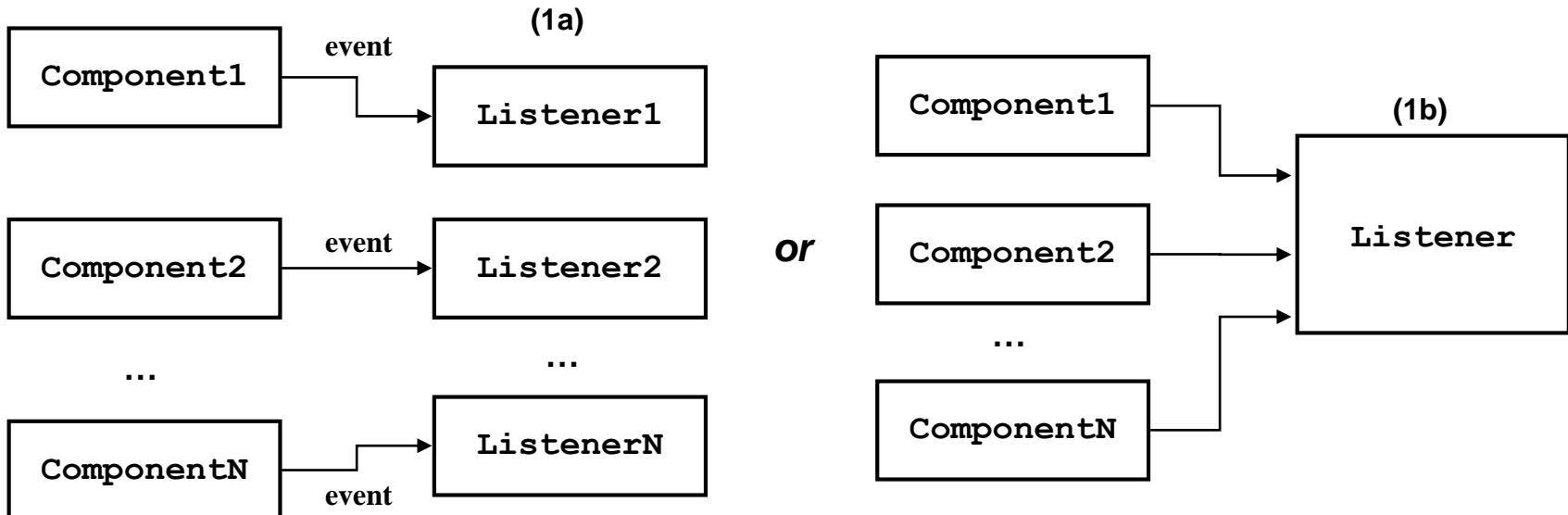
        // register THIS object (the form) as an Action Listener for
        // action events from the button
        myButton.addActionListener(this);

        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 infoTitleBarAreaItem) 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);
        //add the command to the side menu, this places its side component (check box) in the side menu
        myToolbar.addComponentToSideMenu(checkSideMenuComp);
        //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...

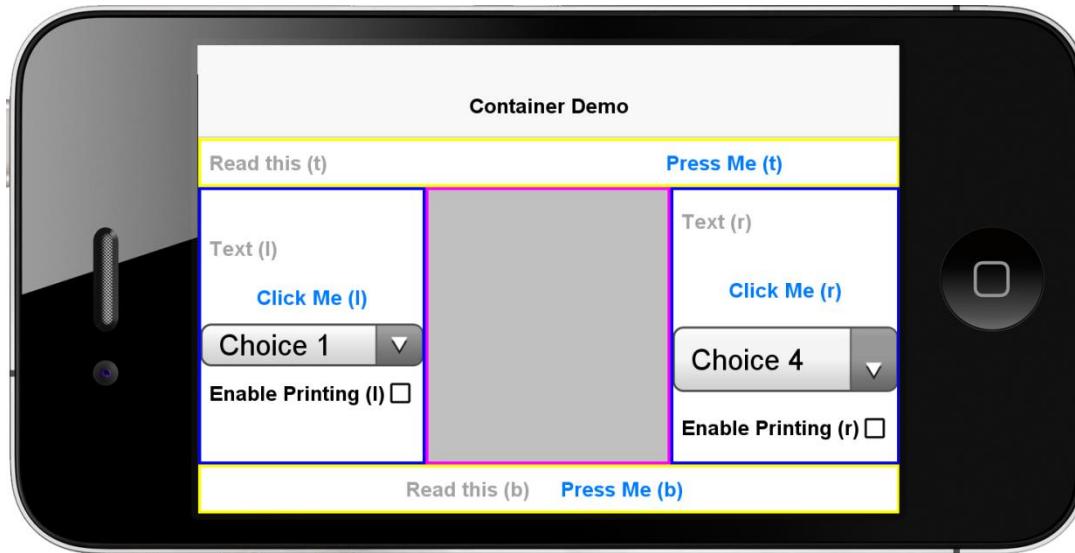
Adding a Command to Side Menu Component

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  
    } //constructor  
} //class
```

Pointer Handling

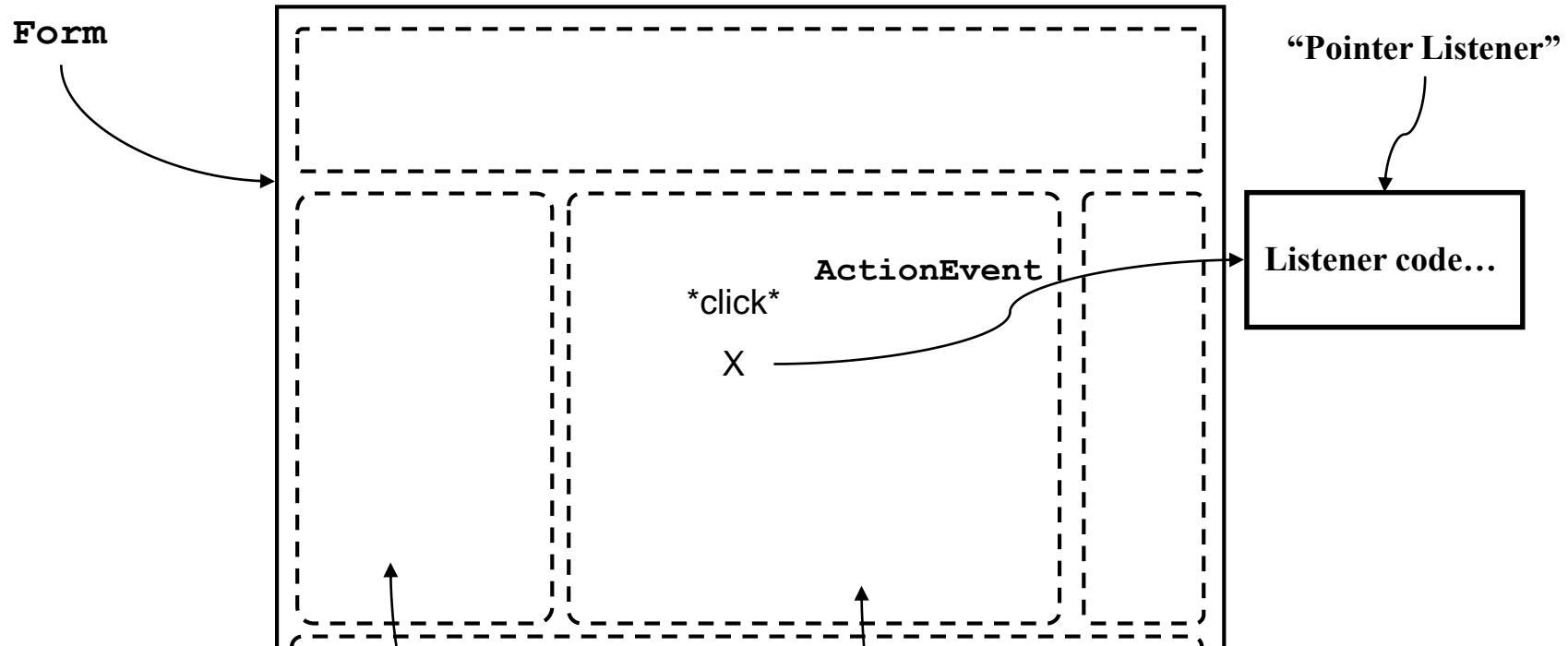
- Components also generate an **ActionEvent** when a pointer is pressed/released or dragged on them.
- **Component** class provides:

```
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) CSC Dept, CSUS

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
 - 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

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

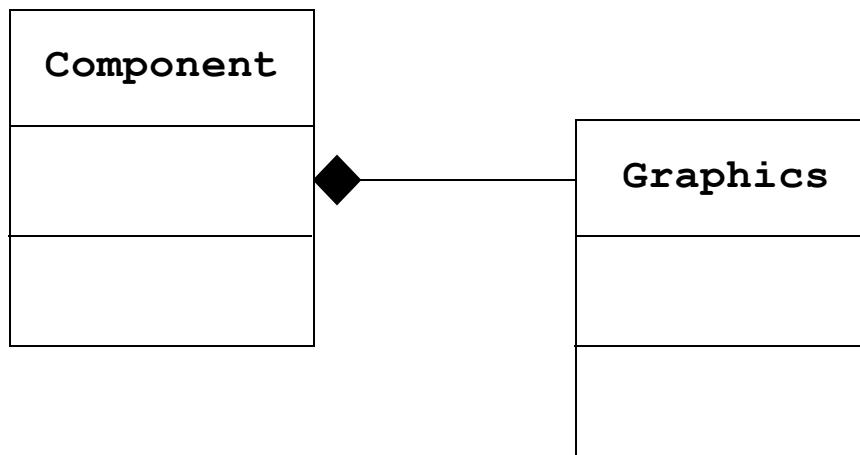
Definition

- An **interaction technique, user interface technique or input technique** is a **combination of hardware and software elements** that provides a way for computer users to accomplish a single task.

Source: https://en.wikipedia.org/wiki/Interaction_technique

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:

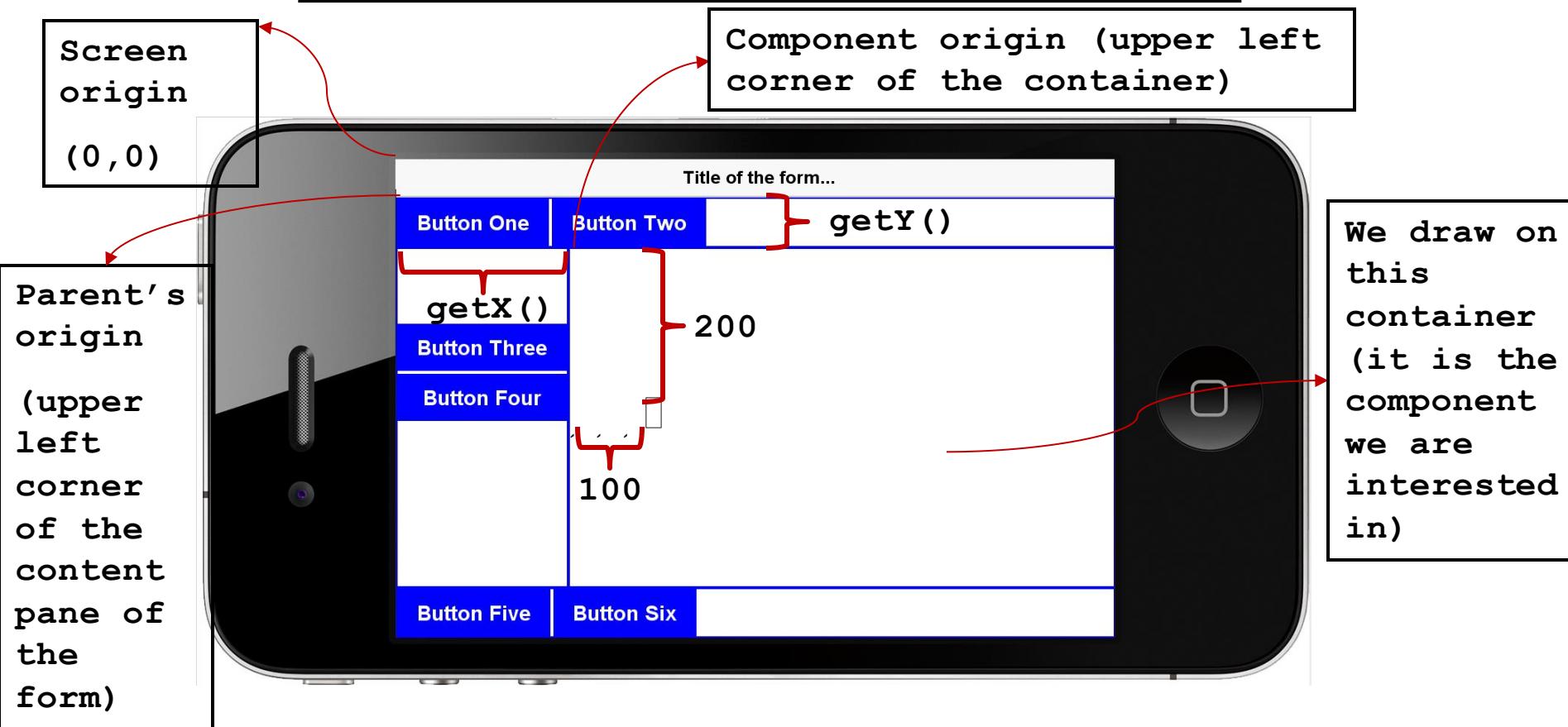
`fillArc(x, y, 2*r, 2*r, 0, 360);`

- `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...

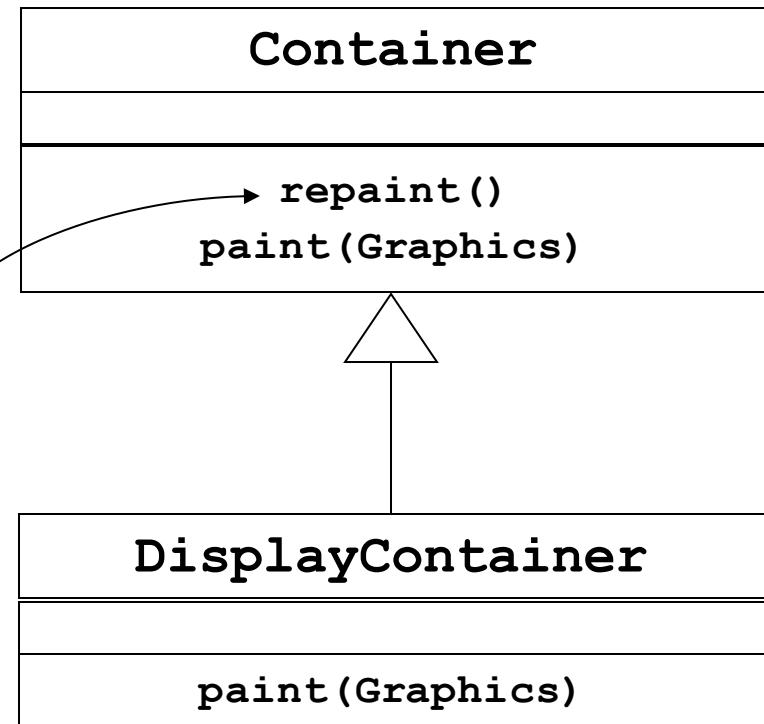
Differences between Java and CN1

- 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...

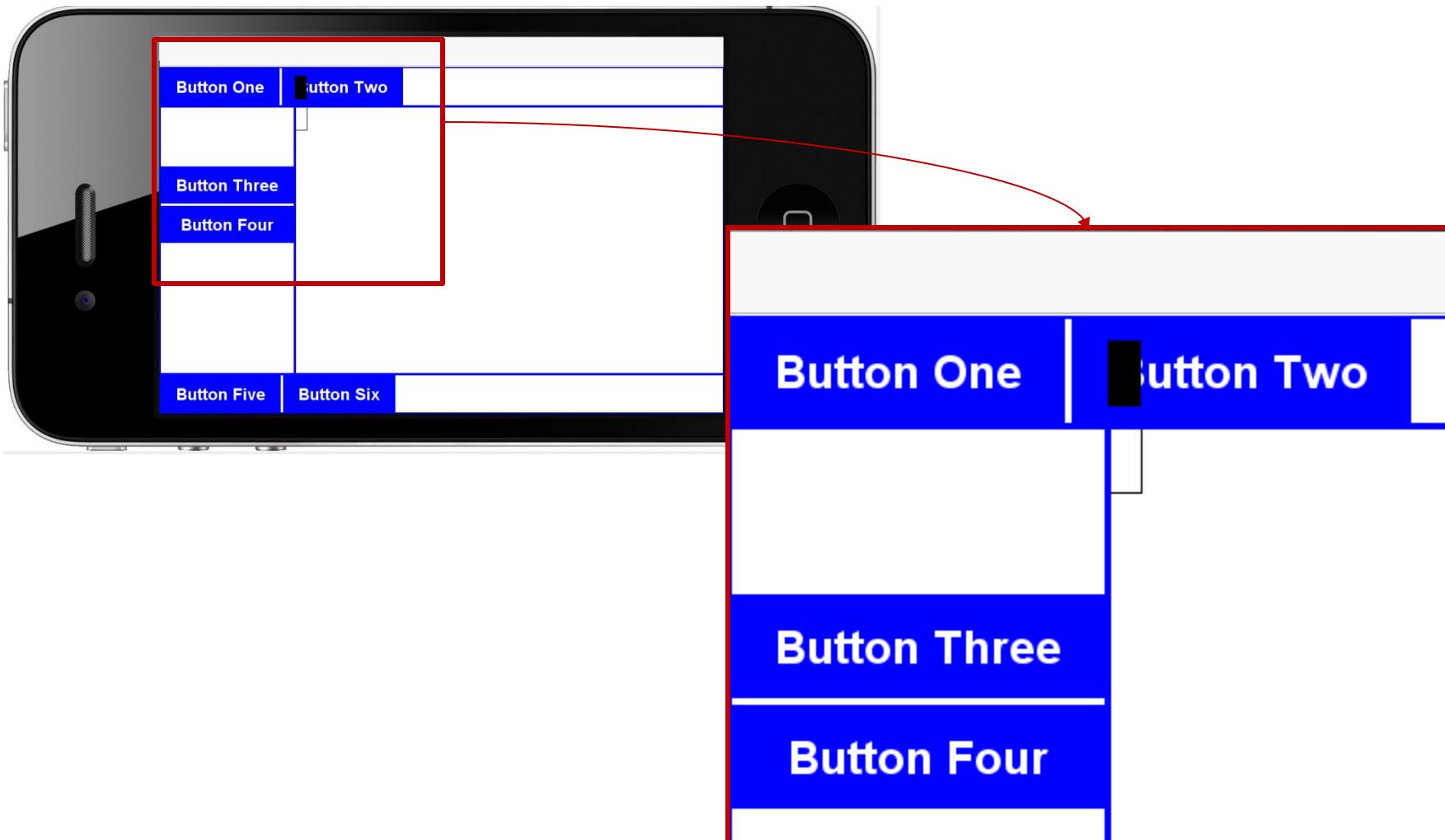
Non-working example

```
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()

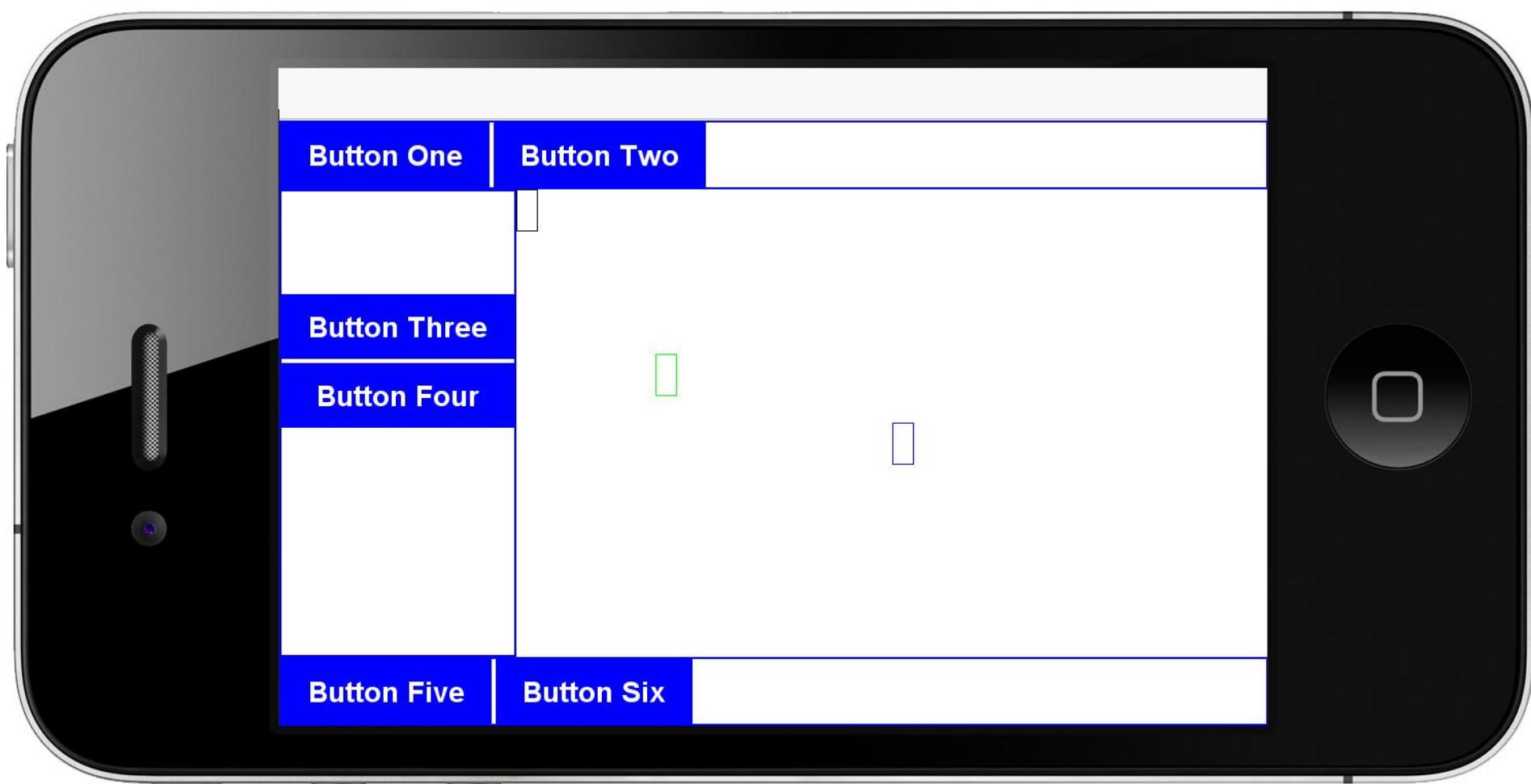
Assume we would like to draw a rectangle in the middle of **CustomContainer**.

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.)

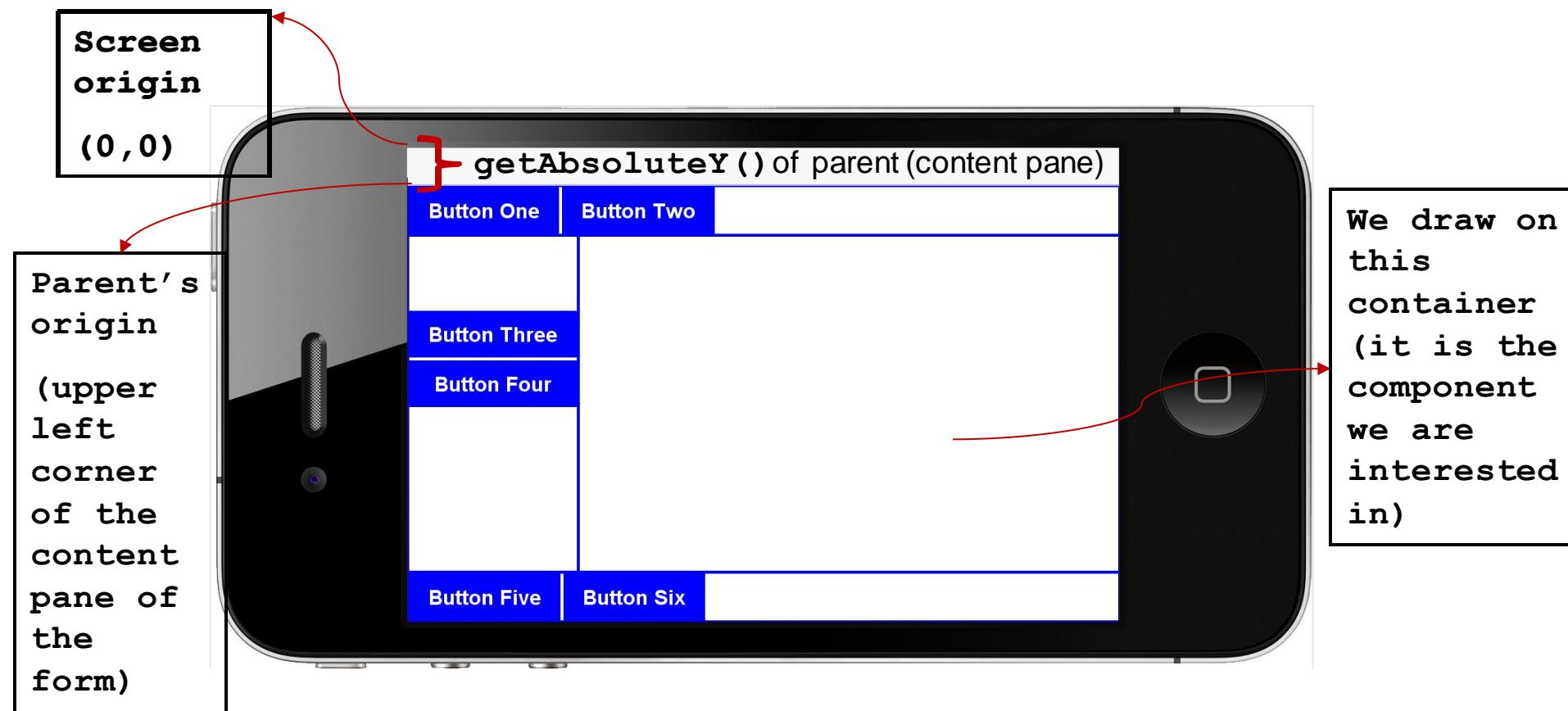
Only the blue rectangle would appear in the center of the **CustomContainer**...



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{

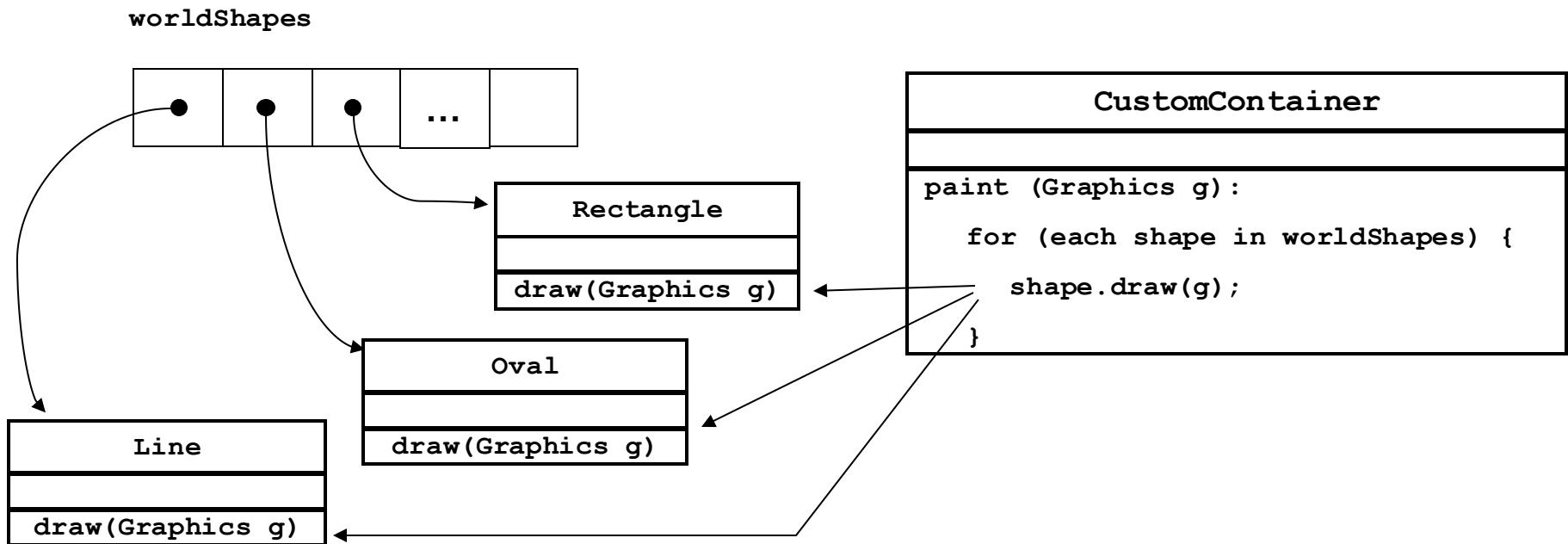
    private int iPx = 0;
    private int iPy = 0;

    @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();
    }
}
```

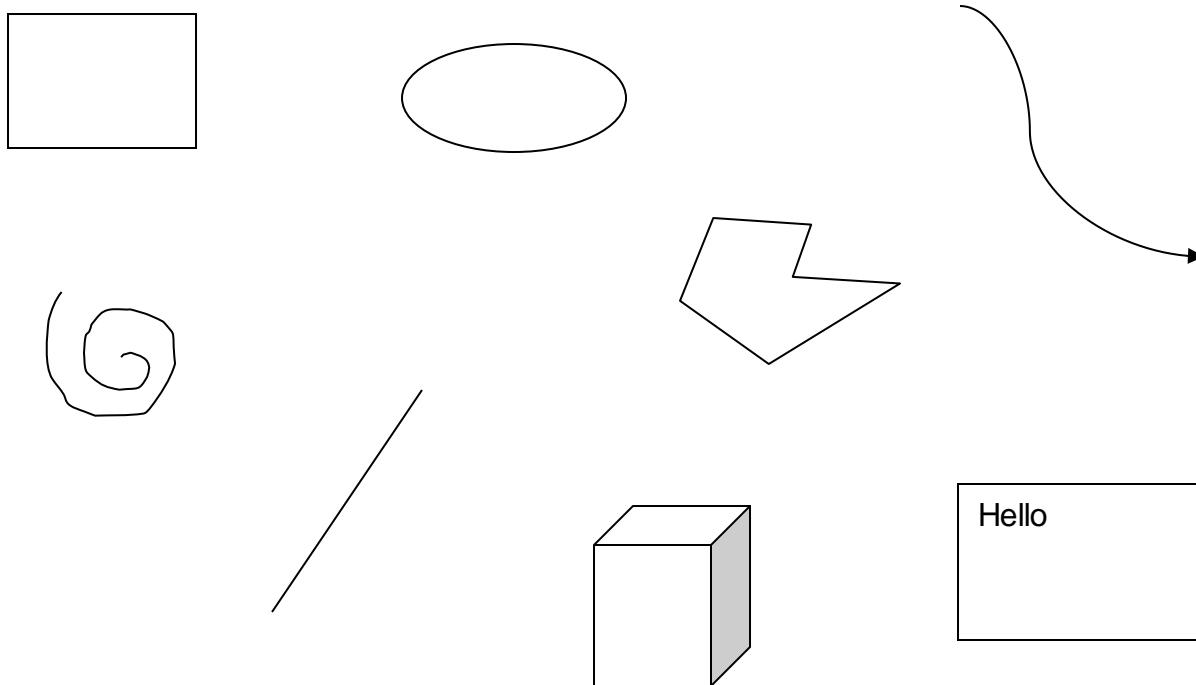
Maintaining Graphical State

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



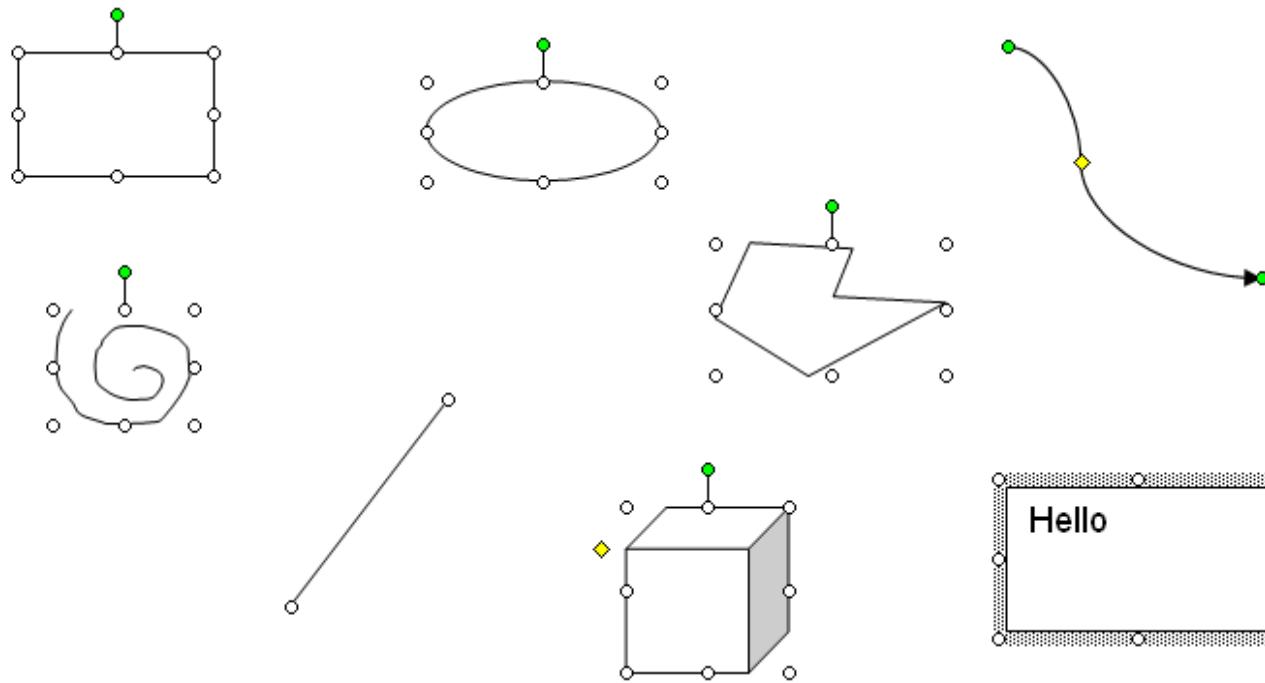
Object Selection

- Various *unselected* objects:



Object Selection (cont.)

- *Selected versions of the same objects:*

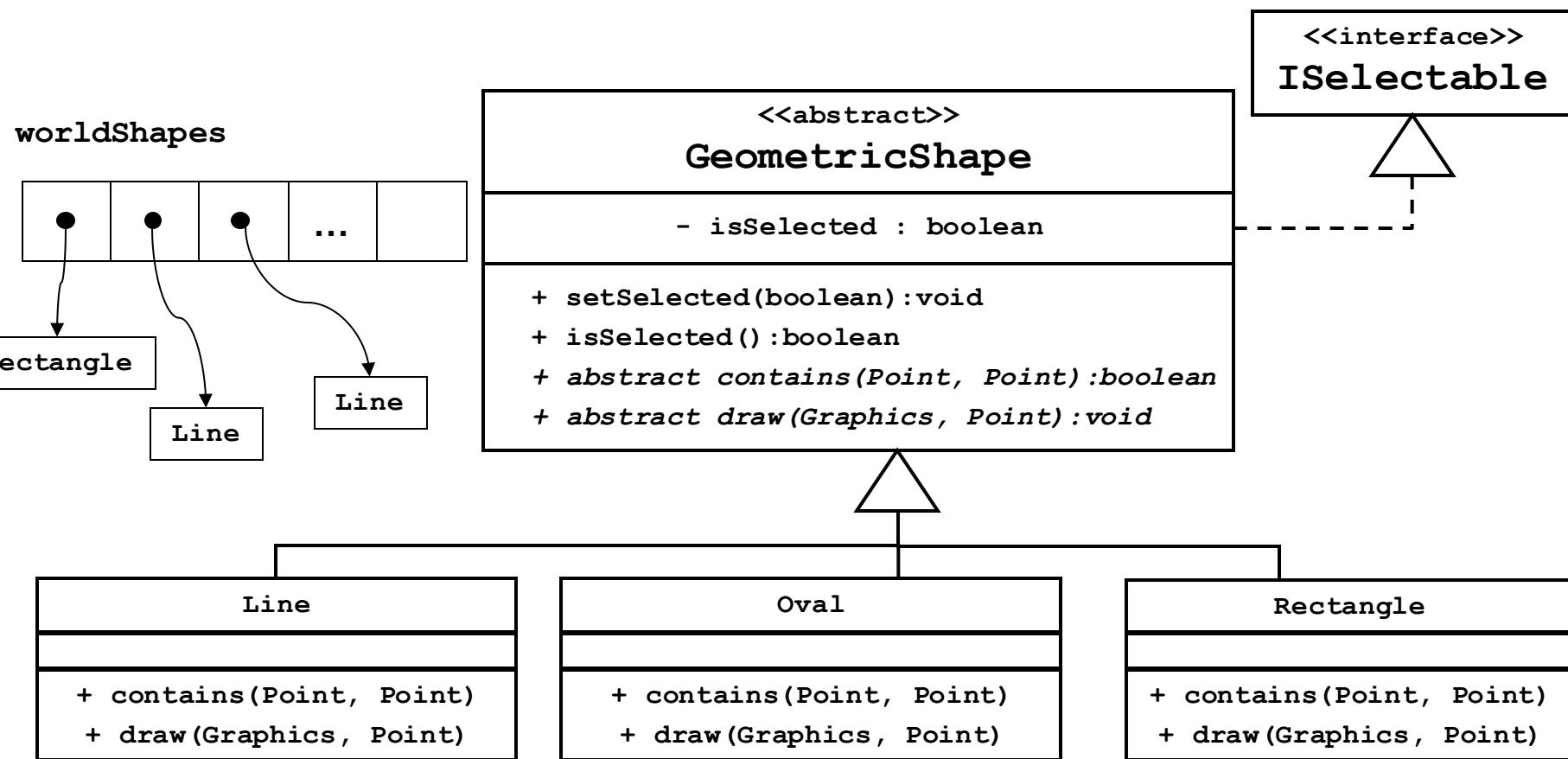


Defining “Selectability”

```
/** 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:

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

```
CustomContainer

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

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

Question: Did you have a feel of how polymorphism takes place here ?

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);}
}

```

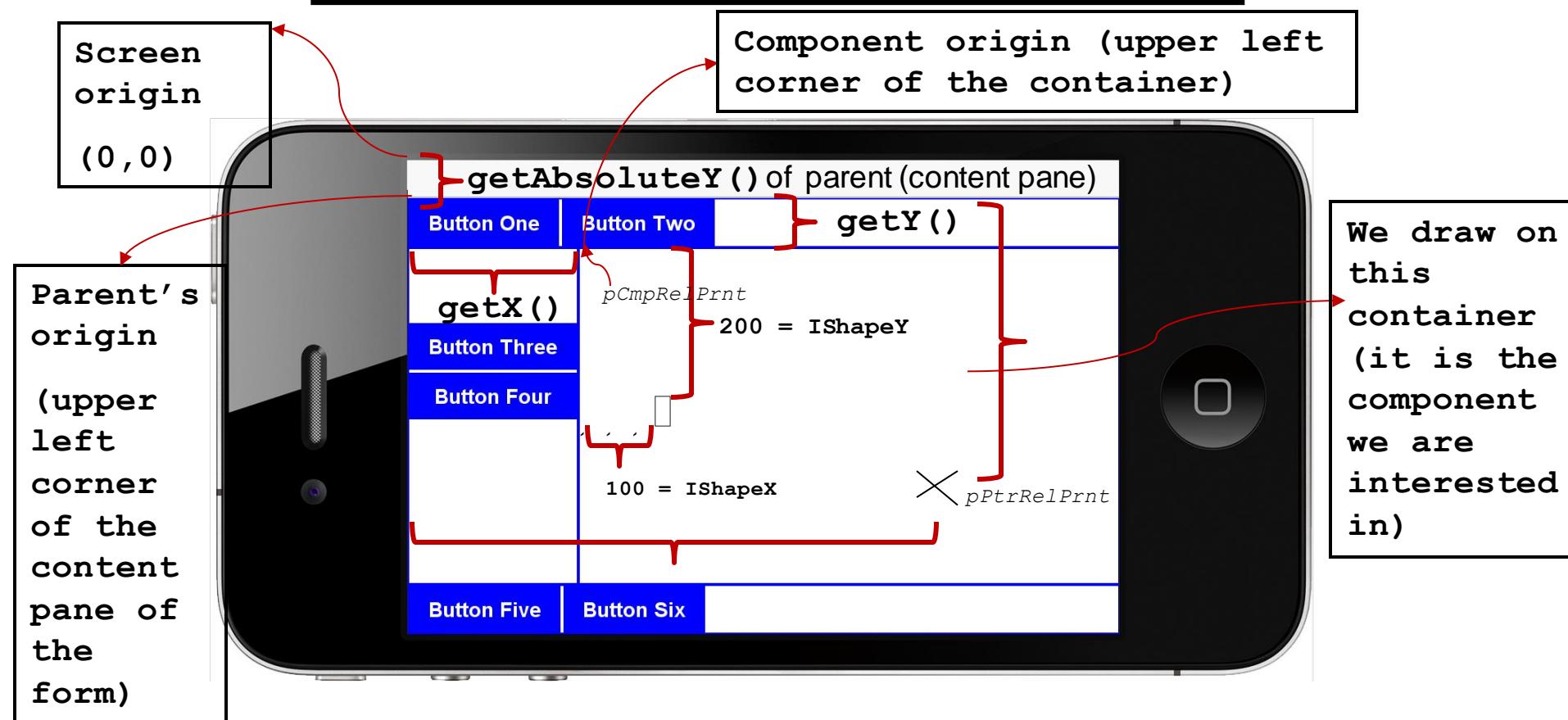
Where is iShapeX & iShapeY define?
How the values get here?

```
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(); } }
```

Backup Slide

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)
```

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

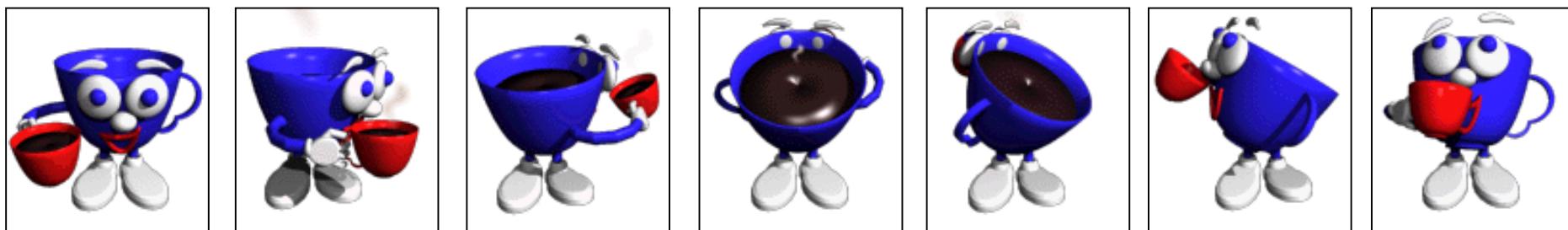
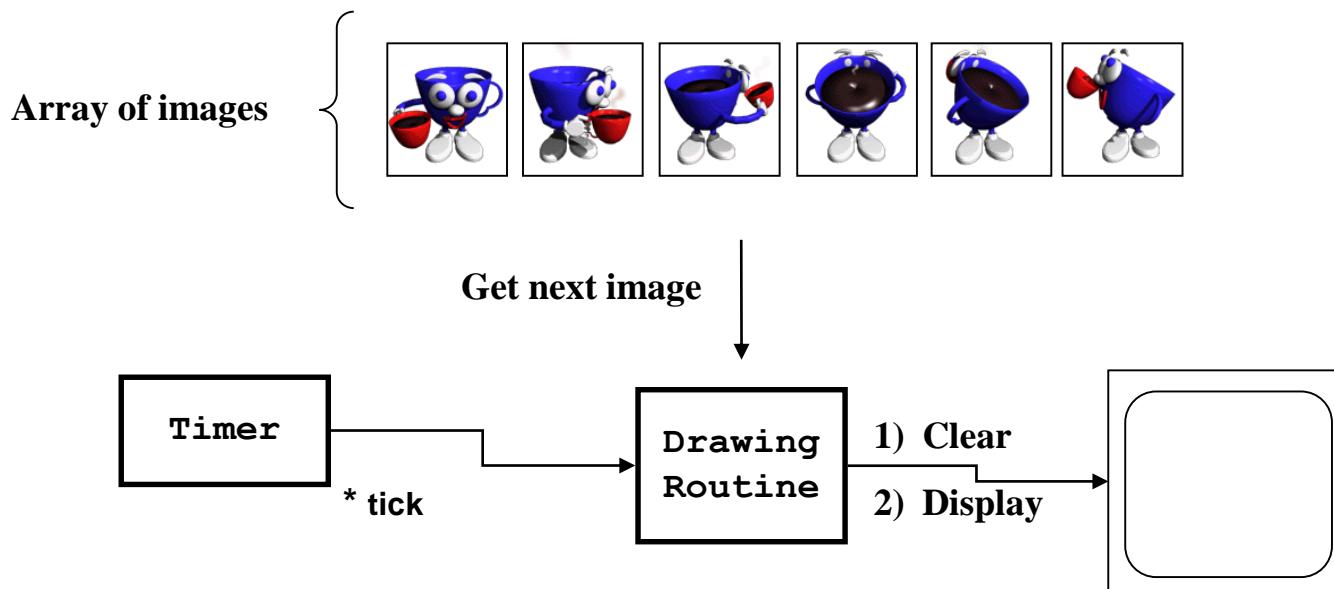


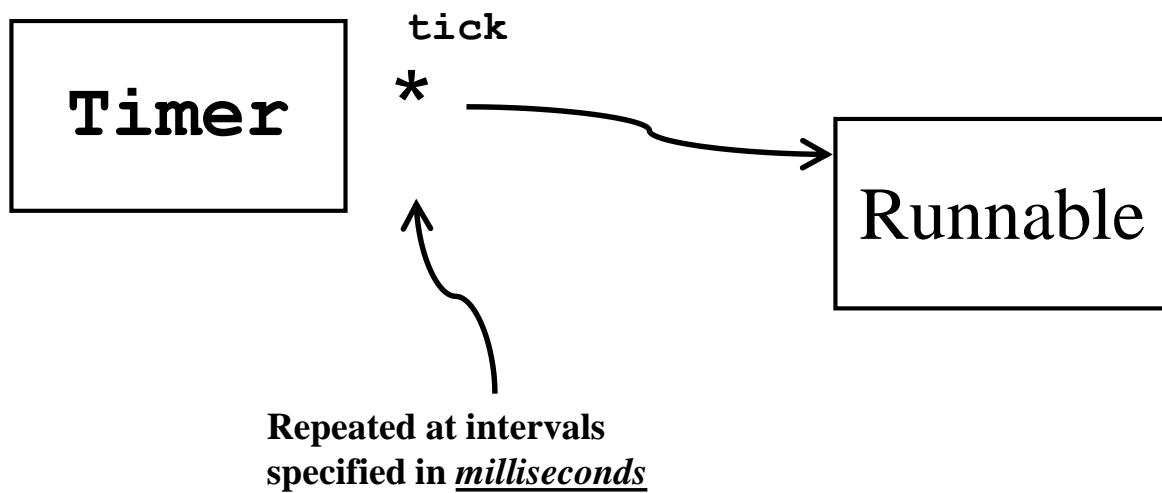
Image credit: [Graphic Java: Mastering the JFC \(3rd ed.\)](#), David Geary

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)

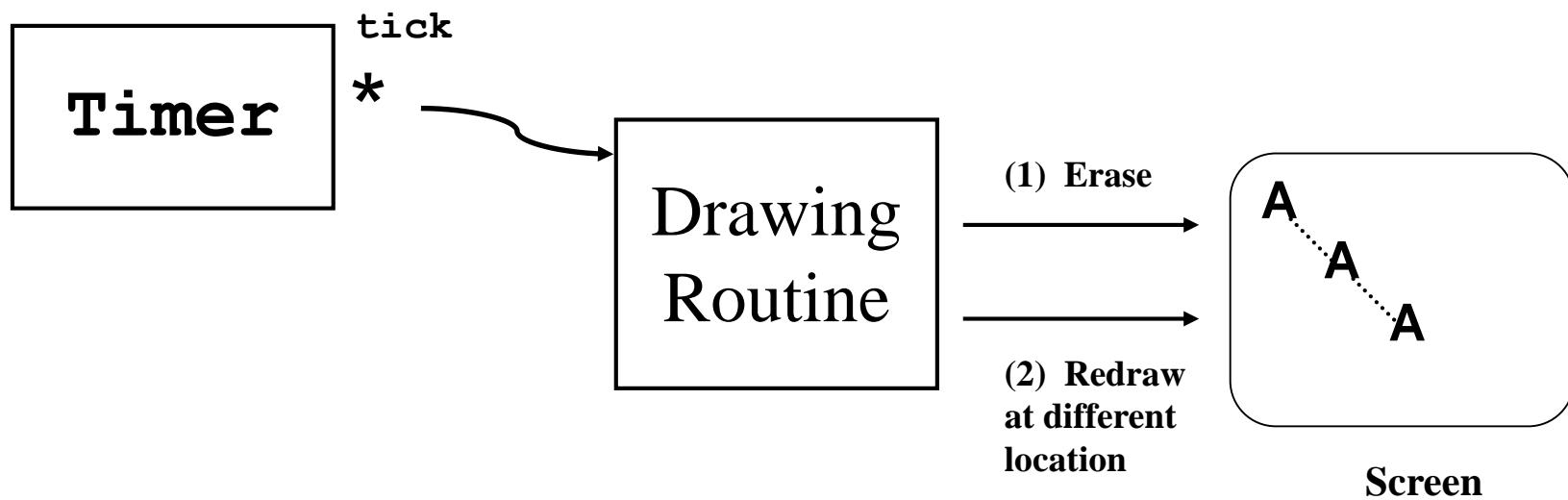
- Runnable attached to the timer must implement interface *Runnable* (build-in CN1 interface):

```
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);
        g.setColor(ColorUtil.BLACK);
        int iShapeX = myRNG.nextInt(getWidth()); //shape location (relative to the
        int iShapeY = myRNG.nextIntgetHeight()); //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

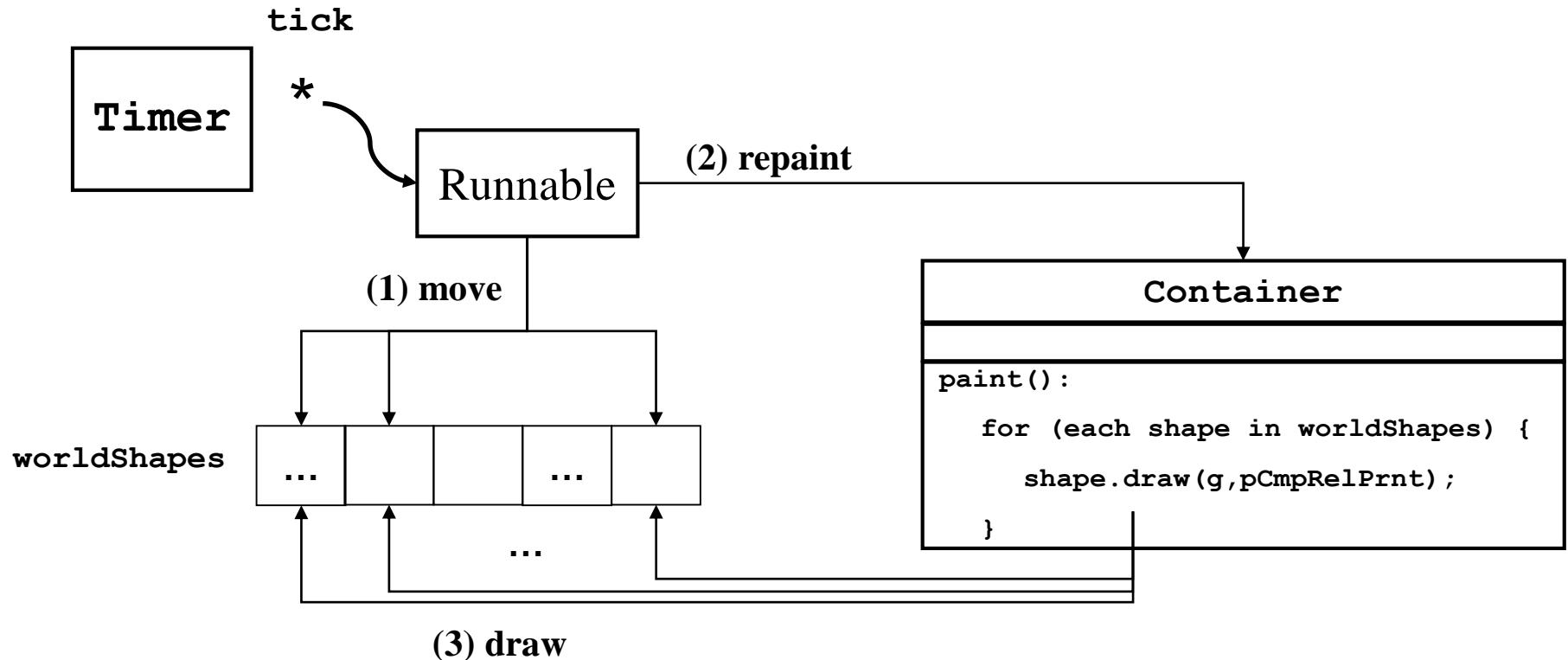


Animation Example

```
/*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 SelfDrawerAnimationForm() {
        //...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 ;
    }
}
```

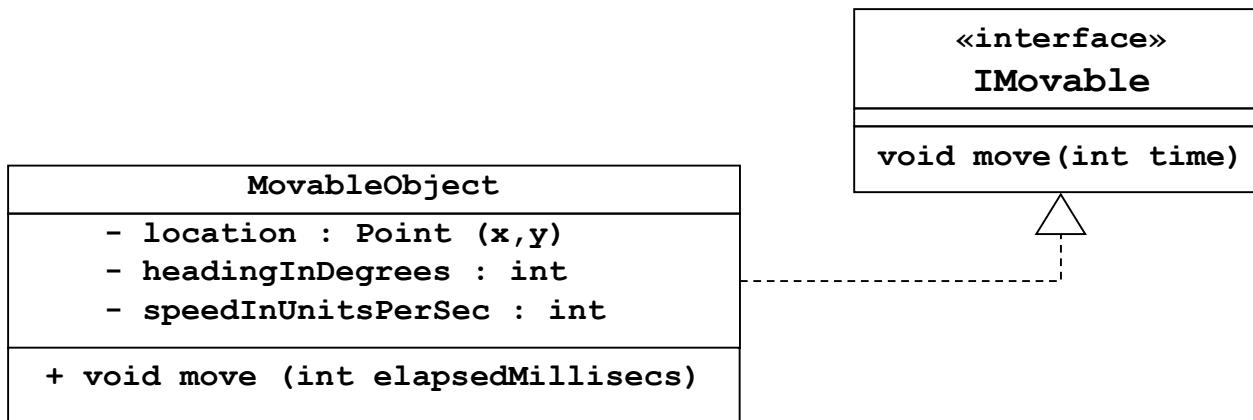
“Self-Animation” Example (cont.)

```
// 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 SelfAnimationContainer (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

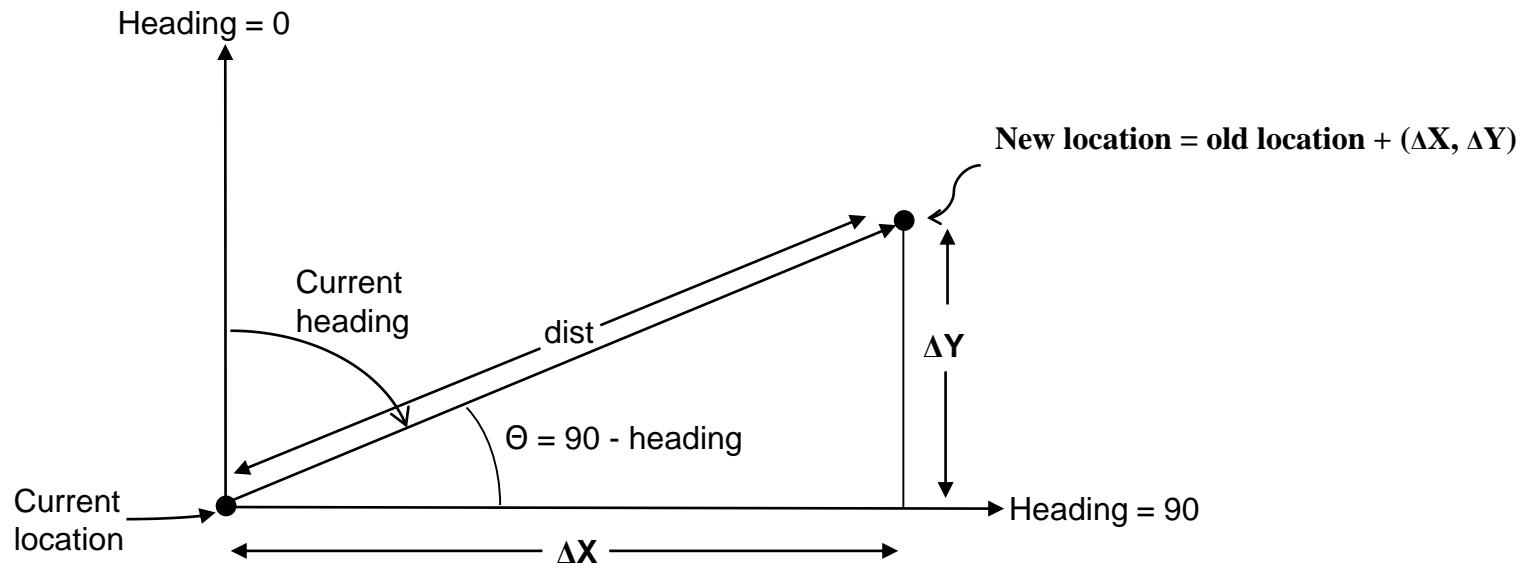
- Consider a “moveable object” defined as:



- 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:



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

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

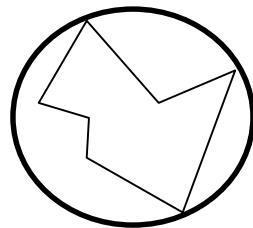
Collision Detection

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

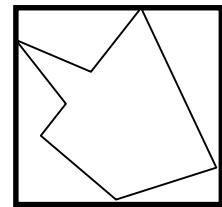
Collision Detection (cont.)

Simplification: “bounding volumes”

- o Areas in the 2D case



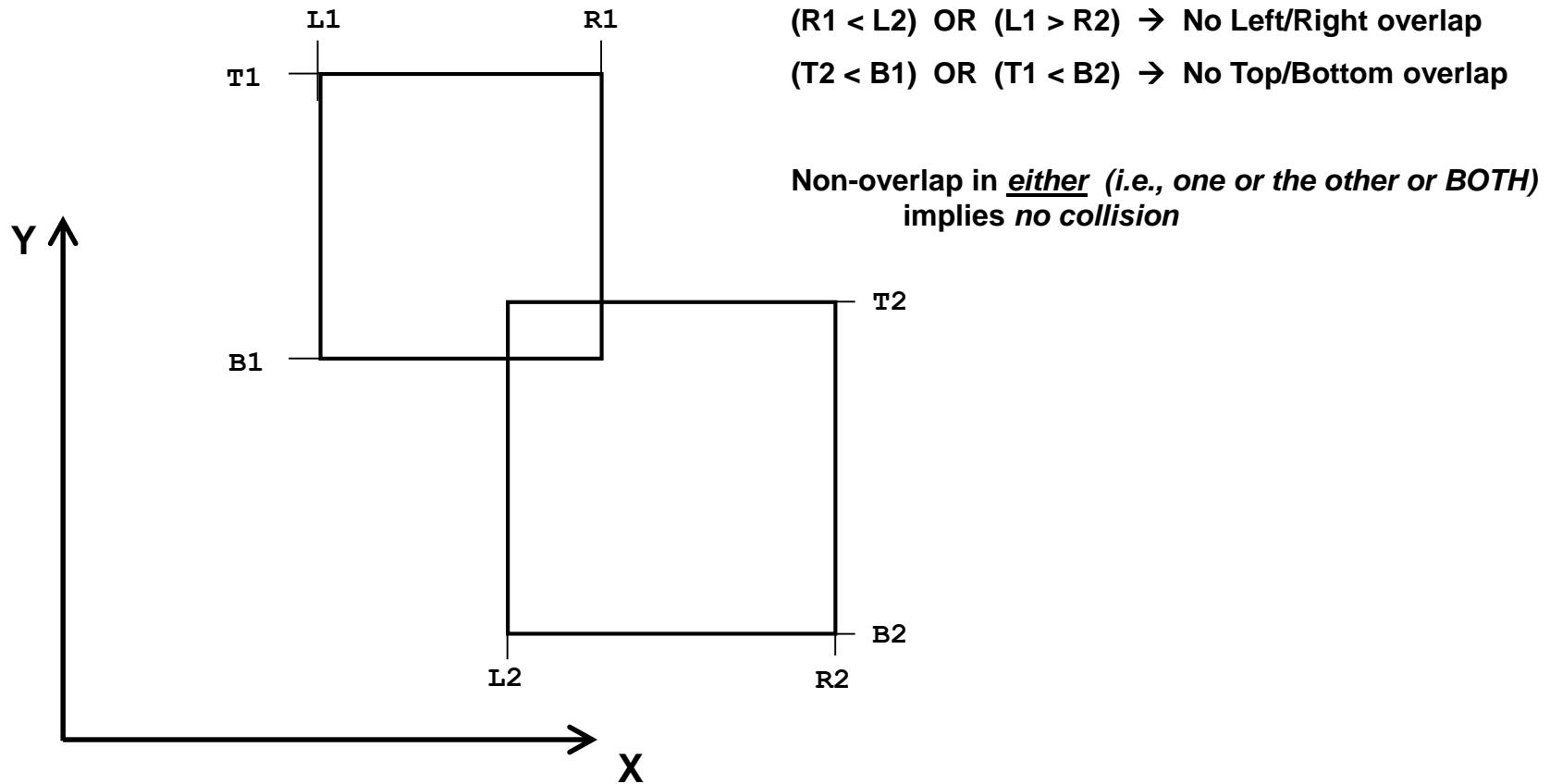
Bounding Circle



Bounding Rectangle

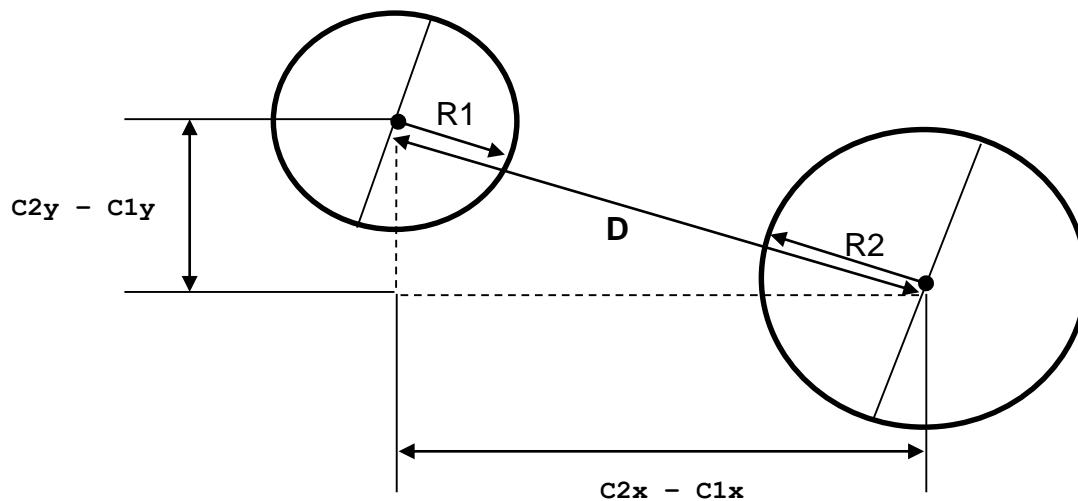
Collision Detection (cont.)

Bounding rectangle collisions



Collision Detection (cont.)

Bounding circle collisions



$$D^2 = (C2_y - C1_y)^2 + (C2_x - C1_x)^2$$

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

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

Collision Response

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

Collision Response (cont.)

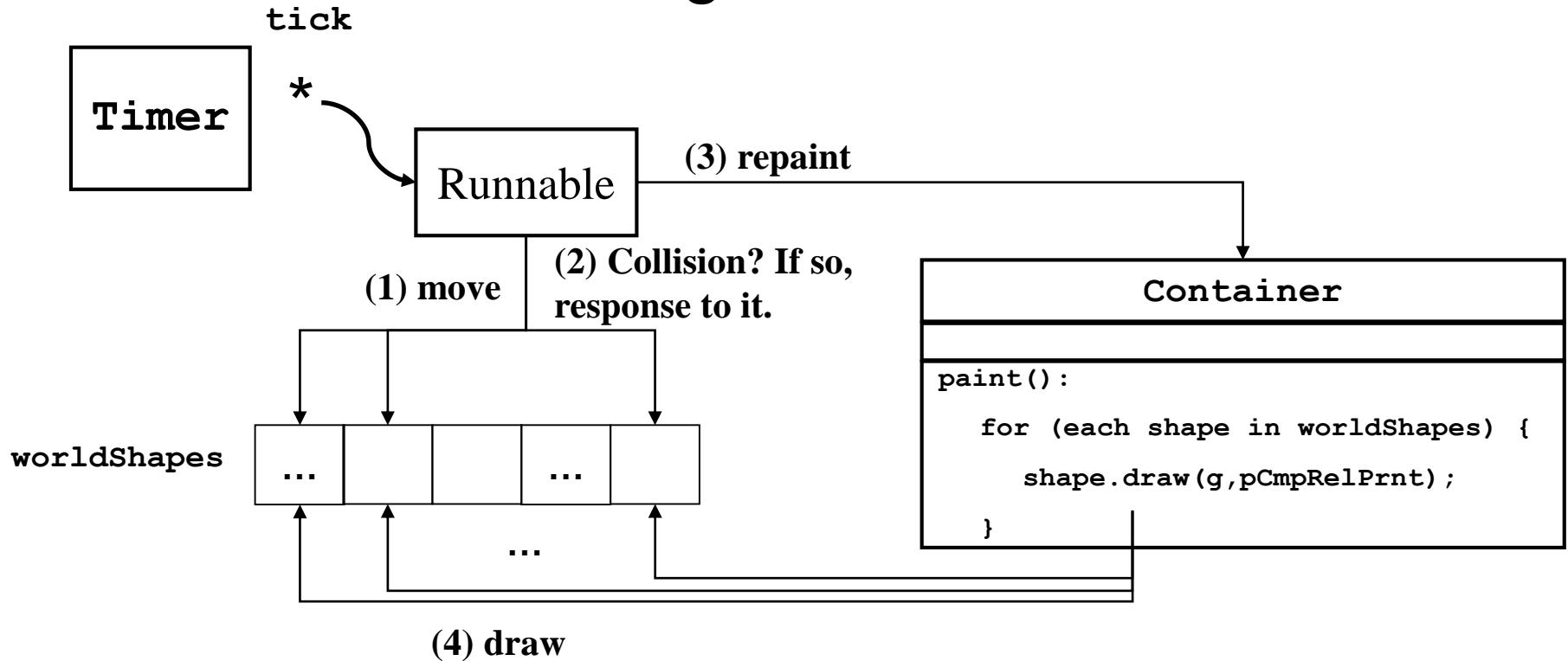
- **Collider interface**

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

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

Handling Collision

- Objects should be responsible for their own drawing, movement, and collision detection/handling.



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 do 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 ;
    }
}
```

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

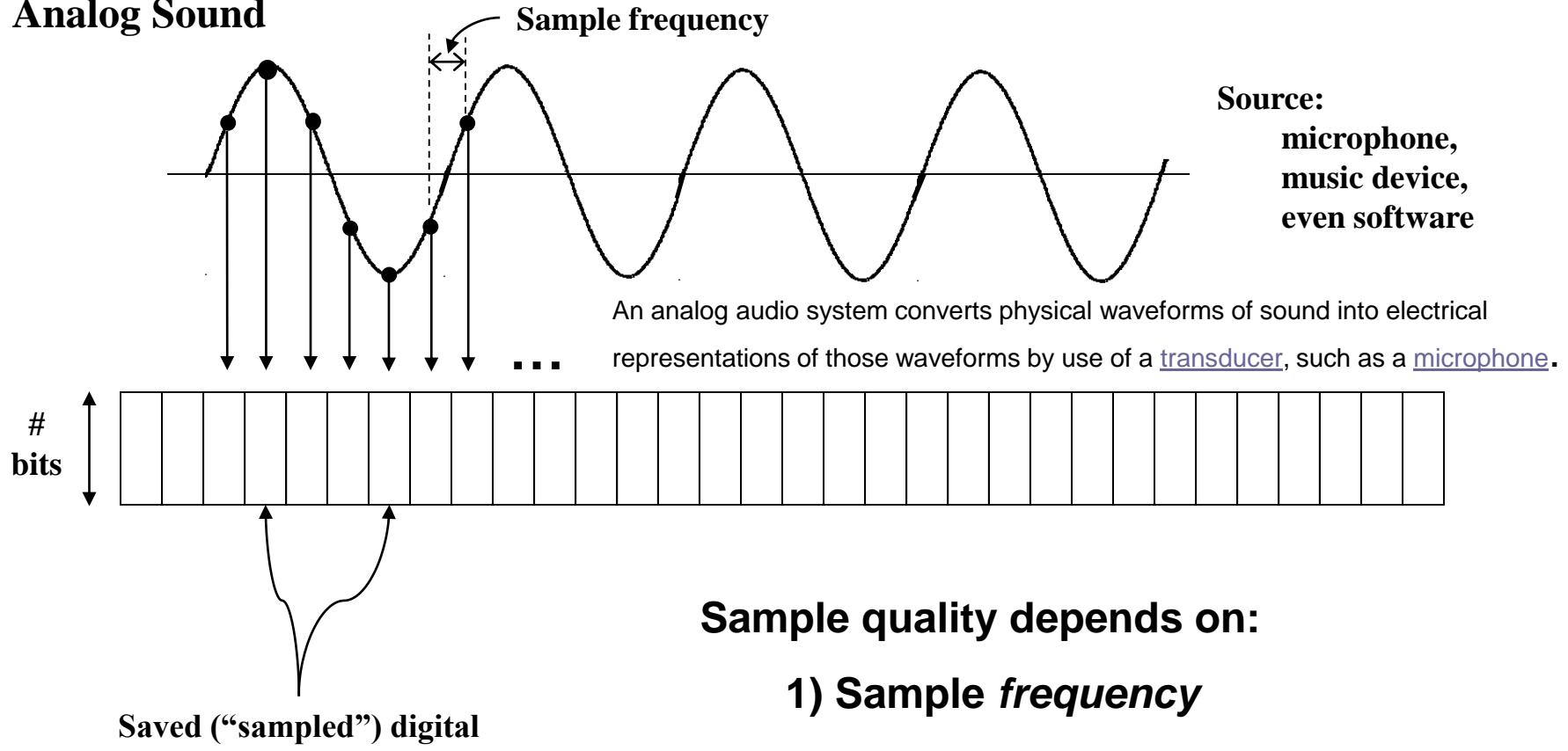
Computer Science Department
California State University, Sacramento

Overview

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

Sampled Audio

Analog Sound

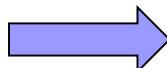


Sound File Formats

- .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)

compressing a sound sequence into a very small file while preserving level of sound quality

- .mid** MIDI file (sequenced, not sampled)
- .ogg** Ogg-Vorbis file (open source)
- .ra** Real Audio (designed for streaming)
- .wav** Windows “wave file”



Finding sound files: www.findsounds.com

Example: WAVE Format

endian	File offset (bytes)	field name	Field Size (bytes)	
big	0	ChunkID	4	The "RIFF" chunk descriptor (ASCII "RIFF", 0x52494646)
little	4	ChunkSize	4	
big	8	Format	4	
big	12	Subchunk1ID	4	
little	16	Subchunk1 Size	4	
little	20	AudioFormat	2	
little	22	NumChannels	2	
little	24	SampleRate	4	
little	28	ByteRate	4	
little	32	BlockAlign	2	
little	34	BitsPerSample	2	
big	36	Subchunk2ID	4	
little	40	Subchunk2 Size	4	
little	44	data	Subchunk2Size	The "data" sub-chunk (ASCII "data", 0x64617461) Indicates the size of the sound information and contains the raw sound data

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

- Import:

```
com.codename1.media.Media;
```

```
com.codename1.media.MediaManager;
```

- **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{

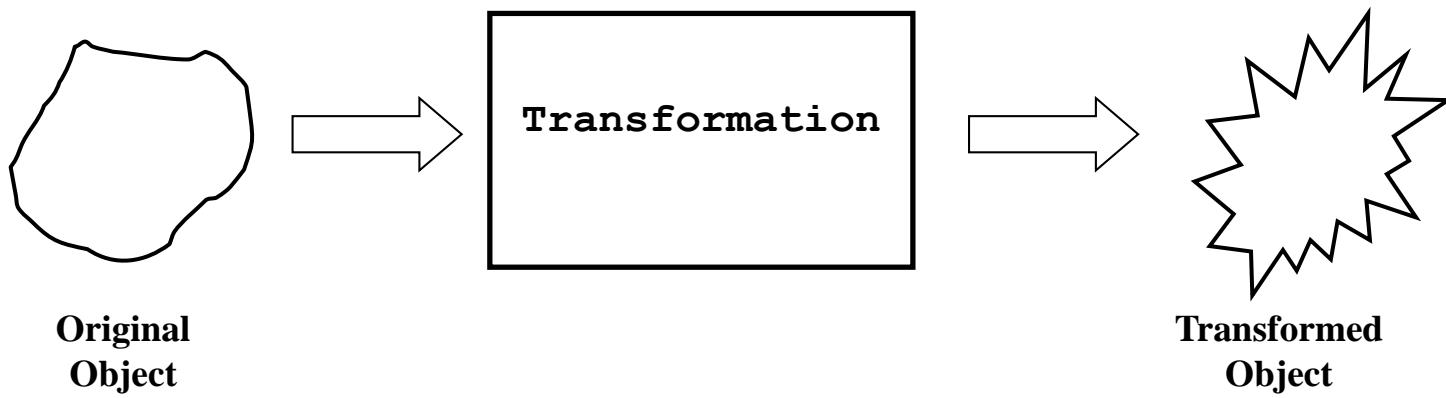
    private BGSound bgSound;
    private boolean bPause = false;
    public BGSoundForm() {
        Button bButton = new Button("Pause/Play");
        //...[style and add bButton to the form]
        bButton.addActionListener(this);
        bgSound = new BGSound("alarm.wav");
        bgSound.play();
   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

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

13 - Transformations

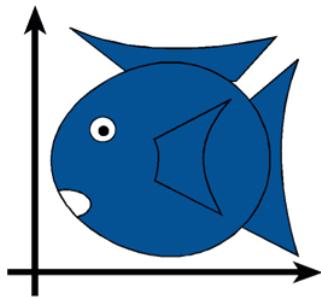
Computer Science Department
California State University, Sacramento

The “Transformation” Concept

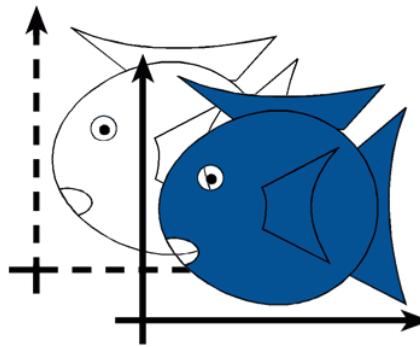


- “Original object” could be anything
 - We will focus on geometric objects
- “Transformed object” is usually (*but not necessarily*) of same type

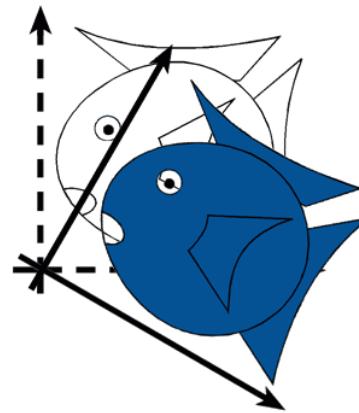
Simple Transformations



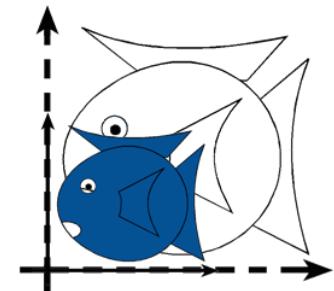
Identity



Translation



Rotation



Isotropic
(Uniform)
Scaling

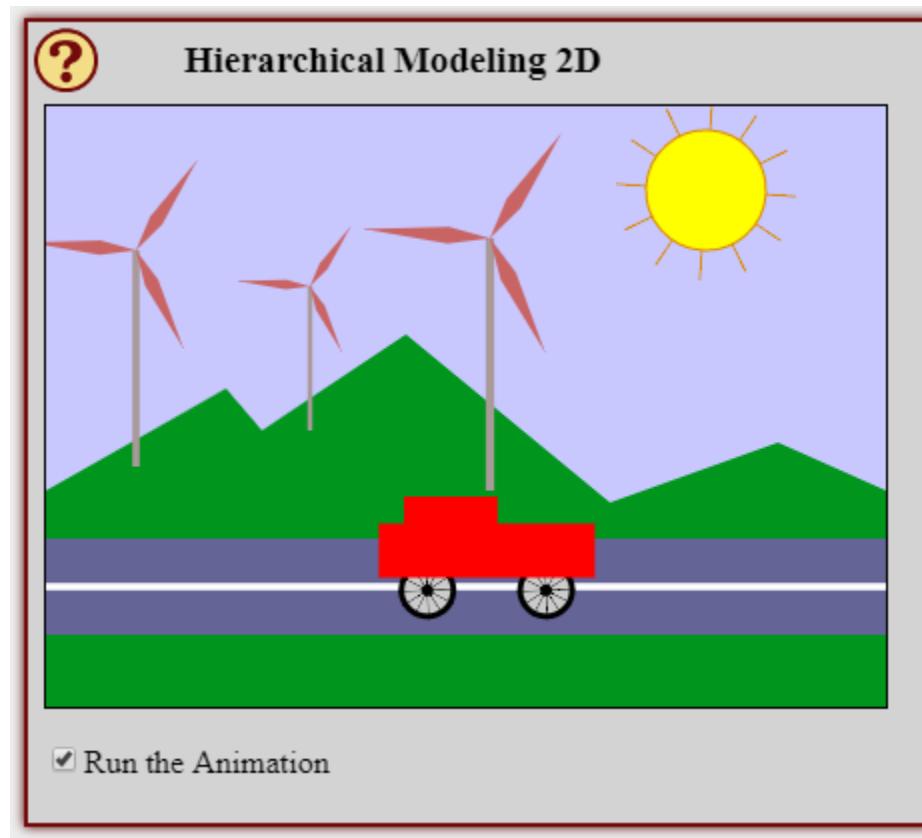
- Can be combined
- Are these operations invertible?

Yes, except scale = 0

Transformations are used:

- Position objects in a scene (modelling)
- Change the shape of objects
- Create multiple copies of objects
- Projection for virtual cameras
- Animations

Example: an animated scene



You can probably guess how hierarchical modeling is used to draw the three windmills in this example. There is a *drawWindmill* method that draws a windmill in its own coordinate system. Each of the windmills in the scene is then produced by applying a different modeling transform to the standard windmill. Furthermore, the windmill is itself a complex object that is constructed from several sub-objects using various modeling transformations.

Overview

- **Part 1:** translate, rotate scale and reflect objects using matrices
(mathematics background - Homework)
- **<https://www.youtube.com/watch?v=DD70ZIDjL7g>**

Overview

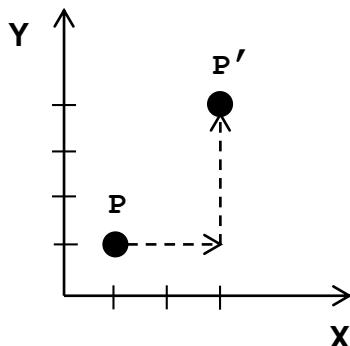
- Part 2:
- **Affine Transformations:** Translation, Rotation, Scaling
- **Transforming Points & Lines**
- **Matrix Representation of Transforms**
- **Homogeneous Coordinates**
- **Concatenation of Transformations**

“Affine” Transformations

- Properties:
 - “Map” (transform) finite points into finite points
 - Map parallel lines into parallel lines
- Common examples used in graphics:
 - Translation
 - Rotation
 - Scaling

Transformations on Points

- Translation



$$P = (x, y)$$

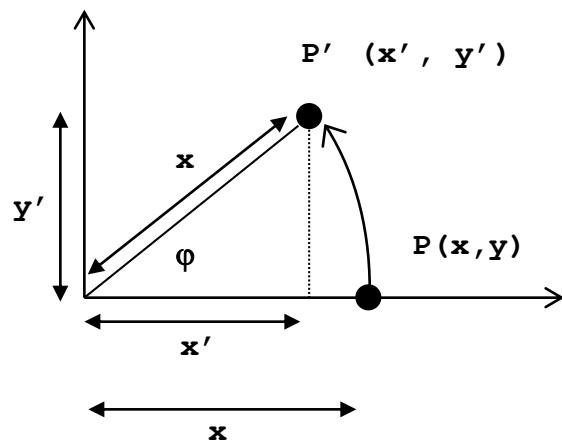
$$T = (+2, +3)$$

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

$$P \rightarrow \boxed{T} \rightarrow P'$$
 or
$$P' \leftarrow \boxed{T} \leftarrow P$$

Transformations on Points (cont.)

- Rotation about the origin (point on X axis)



$$\cos(\varphi) = x' / x ; \text{ hence}$$

$$x' = x \cos(\varphi)$$

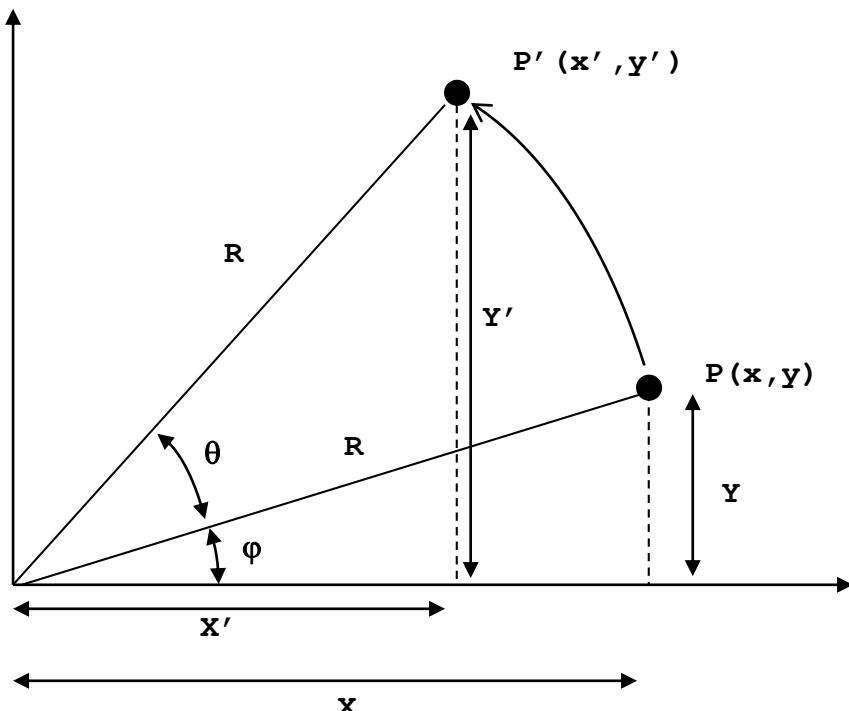
$$\sin(\varphi) = y' / x ; \text{ hence}$$

$$y' = x \sin(\varphi)$$

$$P \rightarrow \boxed{R} \rightarrow P' \quad \text{or} \quad P' \leftarrow \boxed{R} \leftarrow P$$

Transformations on Points (cont.)

- Rotation about the origin (arbitrary point)



$$\cos(\phi) = x / R \text{ and } \sin(\phi) = y / R;$$

$$x = R \cos(\phi) \text{ and } y = R \sin(\phi)$$

$$x' = R \cos(\phi + \theta)$$

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

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

$$= \underline{x} \cos(\theta) - \underline{y} \sin(\theta)$$

Similarly,

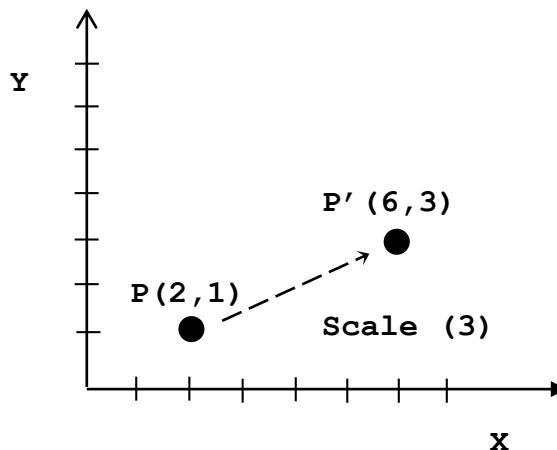
$$y' = x \sin(\theta) + y \cos(\theta)$$

Sum-Difference Formulas

$$\begin{aligned}\sin(u \pm v) &= \sin u \cos v \pm \cos u \sin v \\ \cos(u \pm v) &= \cos u \cos v \mp \sin u \sin v\end{aligned}$$

Transformations on Points (cont.)

- Scaling
 - Multiplication by a “scale factor”



$$P = (x, y)$$

$$S = (s_x, s_y)$$

$$P' = (x * s_x, y * s_y)$$

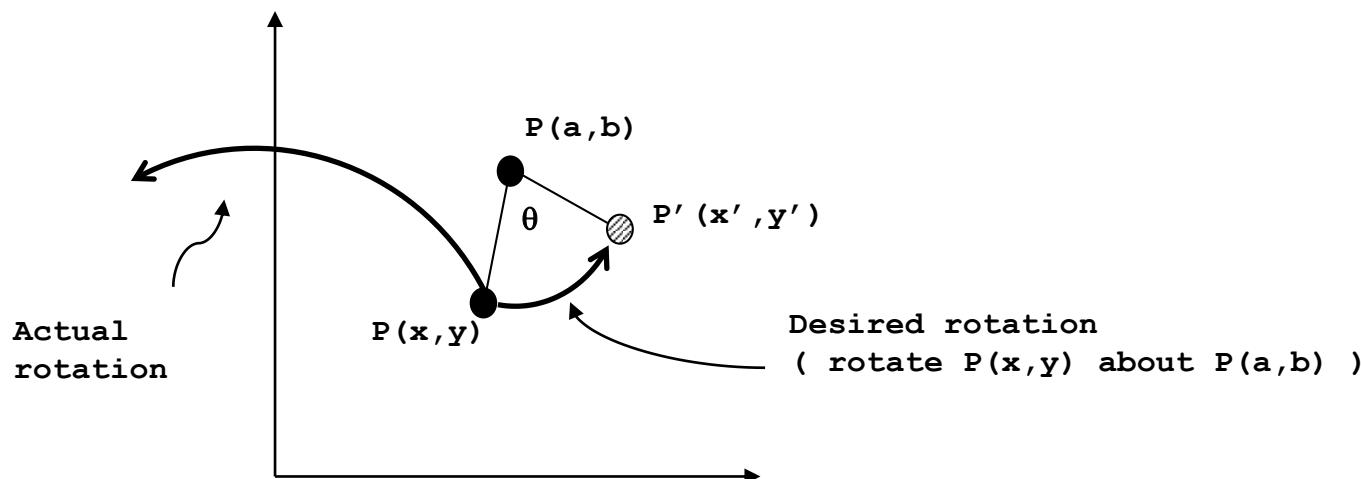
$$P \rightarrow \boxed{S} \rightarrow P' \quad \text{or} \quad P' \leftarrow \boxed{S} \leftarrow P$$

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:
$$P(1,1) * \text{Scale}(3,3) \rightarrow P'(3,3) \quad (\text{"move" of } 2)$$
$$P(4,4) * \text{Scale}(3,3) \rightarrow P'(12,12) \quad (\text{"move" of } 8)$$
- 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”

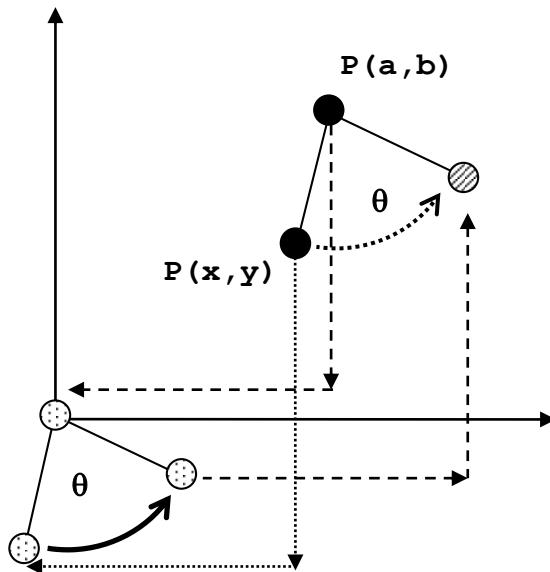
Transformations on Points (cont.)

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



Transformations on Points (cont.)

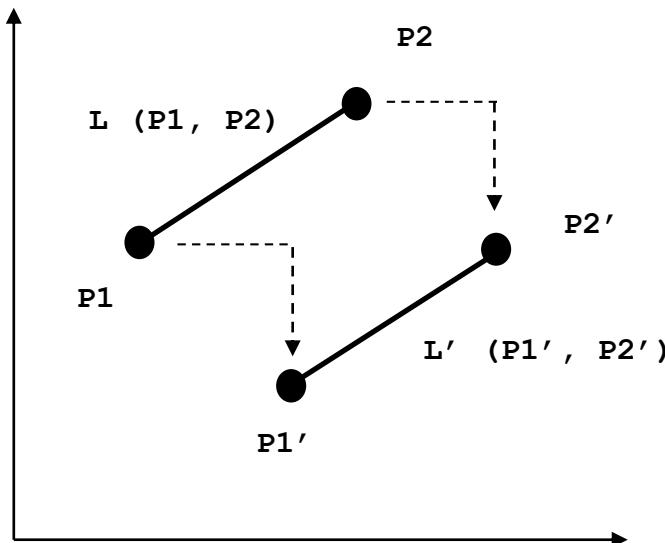
- Solution:
 - Translate to origin
 - Perform rotation
 - Translate “back”



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

Transformations on Lines

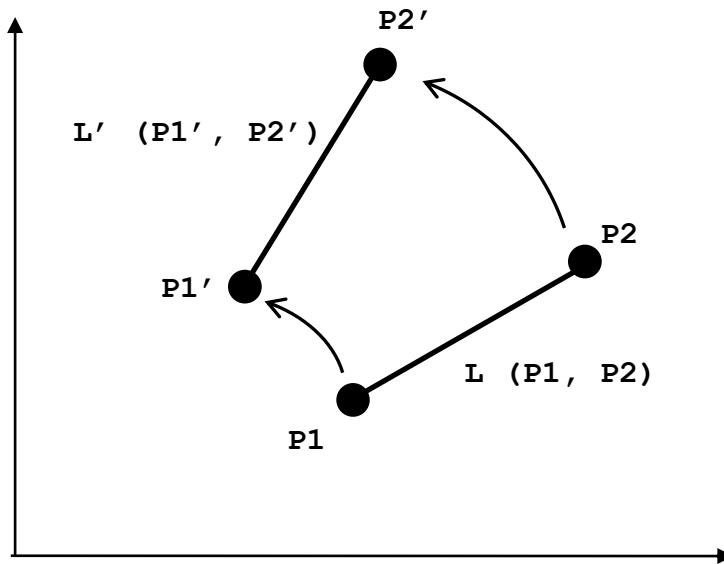
- Translation: translate the endpoints



- **Translate (Line(p_1, p_2))
= Line (Translate(p_1), Translate(p_2))**

Transformations on Lines (cont.)

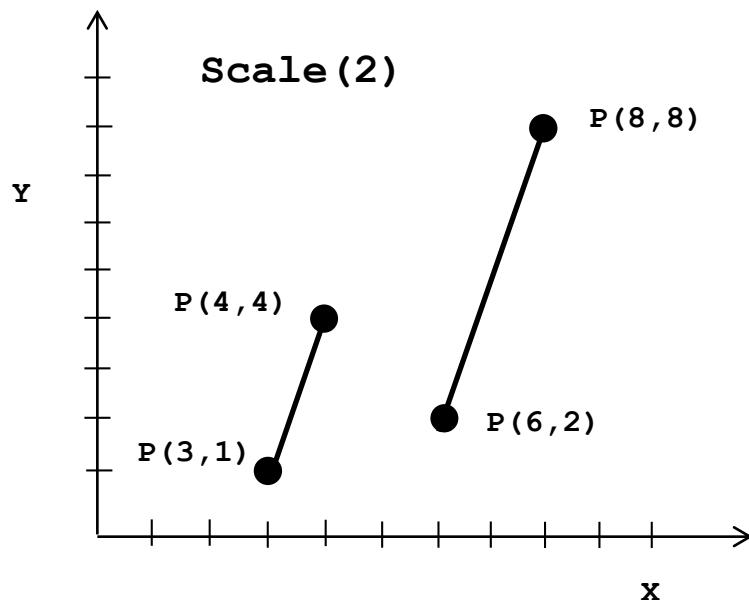
- Rotation about the origin: rotate the endpoints



- **Rotate (Line(p_1, p_2))**
= Line (Rotate(p_1), Rotate(p_2))

Transformations on Lines (cont.)

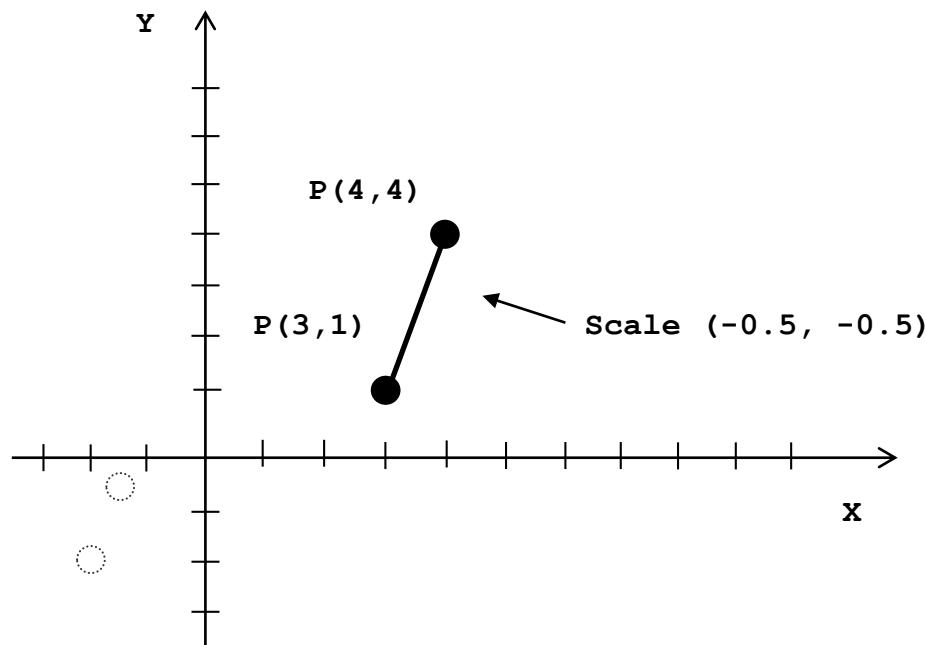
- Scaling: scale the endpoints



- $\text{Scale}(\text{Line}(p_1, p_2)) = \text{Line}(\text{Scale}(p_1), \text{Scale}(p_2))$
- 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:

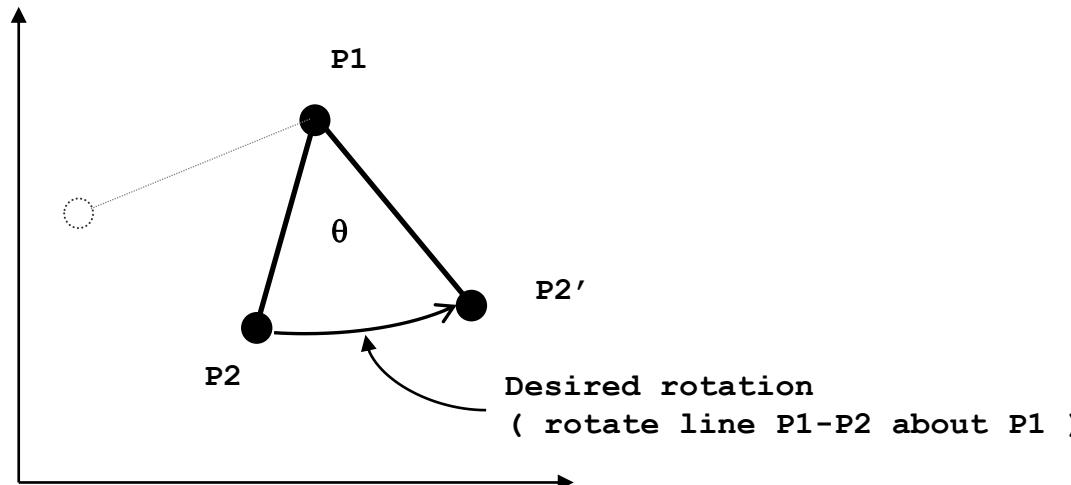
- Absolute Value of Scale Factor $> 1 \rightarrow$ “bigger”
- Absolute Value of Scale Factor $< 1 \rightarrow$ “smaller”
- Scale Factor $< 0 \rightarrow$ “flip” (“mirror”)

Identity Operations:

- For translation: $0 \rightarrow$ No Change
- For rotation: $0 \rightarrow$ No Change
- For scaling: $1 \rightarrow$ No Change

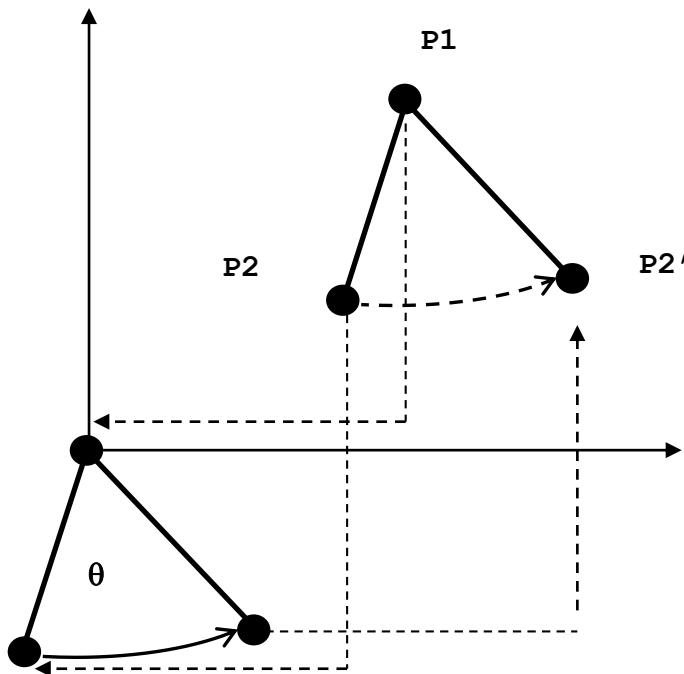
Transformations on Lines (cont.)

- Rotating a line about an endpoint
 - Intent: P_1 doesn't change, while $P_2 \rightarrow P_2'$
(i.e. rotate P_2 by θ about P_1)
 - Again recall: rotation formulas are *about the origin*
 - What is the result of applying *Rotate (θ)* to P_2 ?



Transformations on Lines (cont.)

- Solution: as before – *force the rotation to be “about the origin”*



- P2.translate (-P1.x, -P1.y)**
- P2.rotate (θ)**
- P2.translate (P1.x, P1.y)**



Note “object-oriented” form

Transformations Using Matrices

- Translation

$$\begin{aligned} P &= (x, y) \\ T &= (+2, +3) \\ P' &= (x+2, y+3) \end{aligned}$$

(see slide # 9)

$$P' = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} (x+2) \\ (y+3) \end{bmatrix}$$

Matrix Transformations (cont.)

- Rotation (CCW) about the origin

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

(see slide # 11)

$$\begin{aligned}P' &= [x \quad y] * \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \\&= [(x \cos(\theta) - y \sin(\theta)) \quad (x \sin(\theta) + y \cos(\theta))]\end{aligned}$$

Matrix Transformations (cont.)

- Scaling

$$\begin{aligned} \mathbf{P} &= (\mathbf{x}, \mathbf{y}) \\ \mathbf{S} &= (s_x, s_y) \\ \mathbf{P}' &= (\mathbf{x} * s_x, \mathbf{y} * s_y) \end{aligned}$$

(see slide # 12)

$$\begin{aligned} P' &= [x \quad y] * \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \\ &= [(x * s_x) \quad (y * s_y)] \end{aligned}$$

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 3×3 *matrix*
 - Use matrix ***multiplication*** for ***all*** transformations

Homogeneous coordinates are ubiquitous in computer graphics because they allow common vector operations such as translation, rotation, scaling and perspective projection to be represented as a **matrix** by which the vector is multiplied.

Homogeneous Transformations

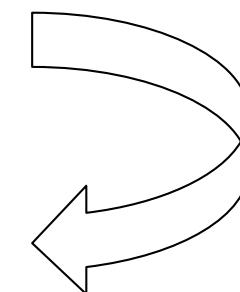
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix}$$

Translation

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

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

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:

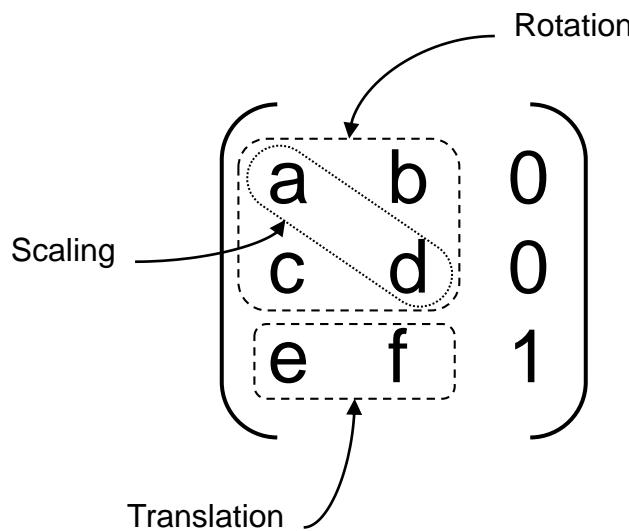
$$\begin{bmatrix} (x\cos(\theta)-y\sin(\theta)) \\ (x\sin(\theta)+y\cos(\theta)) \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Scaling:

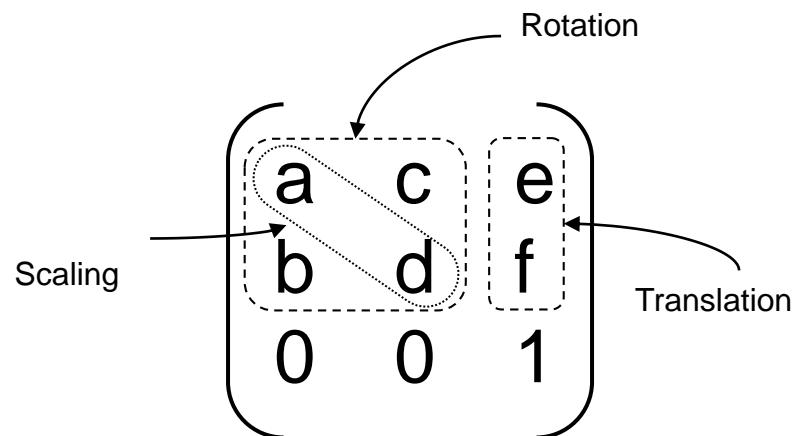
$$\begin{bmatrix} (x*S_x) \\ (y*S_y) \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Active Matrix Areas

Row-major form



Column-major form



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

Technically, these two ways of expressing points and vectors as matrices are perfectly valid and choosing one mode or the other is just a matter of convention.

Vector written as [1x3] matrix: $\mathbf{V} = [x \ y \ z]$

Vector written as [3x1] matrix: $\mathbf{V} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$

Concatenation of Transforms

Typical Sequence:

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

P2 \times **Rotate(θ)** = P3 ;

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

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

Concatenation of Transforms (cont.)

- In (row-major) Matrix Form:

$$\begin{pmatrix} x_1 & y_1 & 1 \end{pmatrix} \times \begin{pmatrix} \text{Translate} \\ (tx, ty) \end{pmatrix} = \begin{pmatrix} x_2 & y_2 & 1 \end{pmatrix}$$

$$\begin{pmatrix} x_2 & y_2 & 1 \end{pmatrix} \times \begin{pmatrix} \text{Rotate}(\theta) \end{pmatrix} = \begin{pmatrix} x_3 & y_3 & 1 \end{pmatrix}$$

$$\begin{pmatrix} x_3 & y_3 & 1 \end{pmatrix} \times \begin{pmatrix} \text{Scale} \\ (sx, sy) \end{pmatrix} = \begin{pmatrix} x_4 & y_4 & 1 \end{pmatrix}$$

$$\begin{pmatrix} x_4 & y_4 & 1 \end{pmatrix} \times \begin{pmatrix} \text{Translate} \\ (tx, ty) \end{pmatrix} = \begin{pmatrix} x_5 & y_5 & 1 \end{pmatrix}$$

Concatenation of Transforms (cont.)

- Alternate Matrix Form:

$$\left(\begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix} \times \begin{bmatrix} T_1 \end{bmatrix} \right) \times \begin{bmatrix} R_1 \end{bmatrix} \times \begin{bmatrix} S_1 \end{bmatrix} \times \begin{bmatrix} T_2 \end{bmatrix}$$
$$= \begin{bmatrix} x_5 & y_5 & 1 \end{bmatrix}$$

Concatenation of Transforms (cont.)

- Matrix multiplication is associative:

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

Note: The **associative** property states that you can add or multiply regardless of how the numbers are grouped.

In Column-Major Form

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

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

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

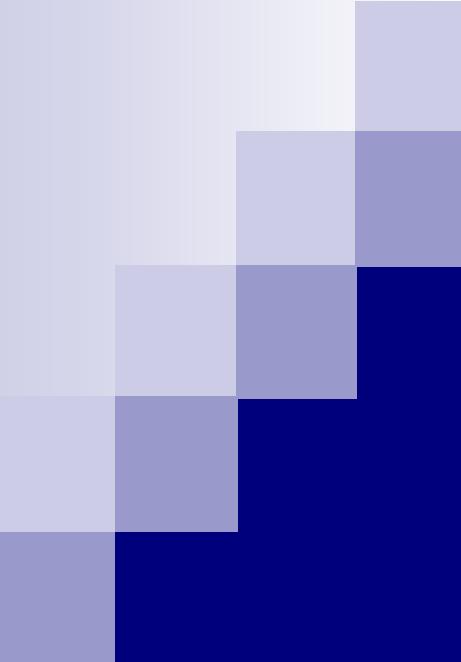
$$\begin{bmatrix} x_5 \\ y_5 \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{Trans} \\ (\mathbf{x}, \mathbf{y}) \end{bmatrix} \times \begin{bmatrix} x_4 \\ y_4 \\ 1 \end{bmatrix}$$

Column-Major Form (cont.)

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

$$\begin{bmatrix} x_5 \\ y_5 \\ 1 \end{bmatrix} = \underbrace{\left[\begin{bmatrix} T2 \end{bmatrix} \times \left[\begin{bmatrix} S1 \end{bmatrix} \times \left[\begin{bmatrix} R1 \end{bmatrix} \times \left[\begin{bmatrix} T1 \end{bmatrix} \right] \right] \right]}_{\text{A large bracket under the first four matrices}} \times \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_5 \\ y_5 \\ 1 \end{bmatrix} = \underbrace{\left[\begin{bmatrix} M \end{bmatrix} \right]}_{\text{A large bracket under the last matrix}} \times \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$



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 2×3 elements can be accessed*

Methods to *manipulate* TM

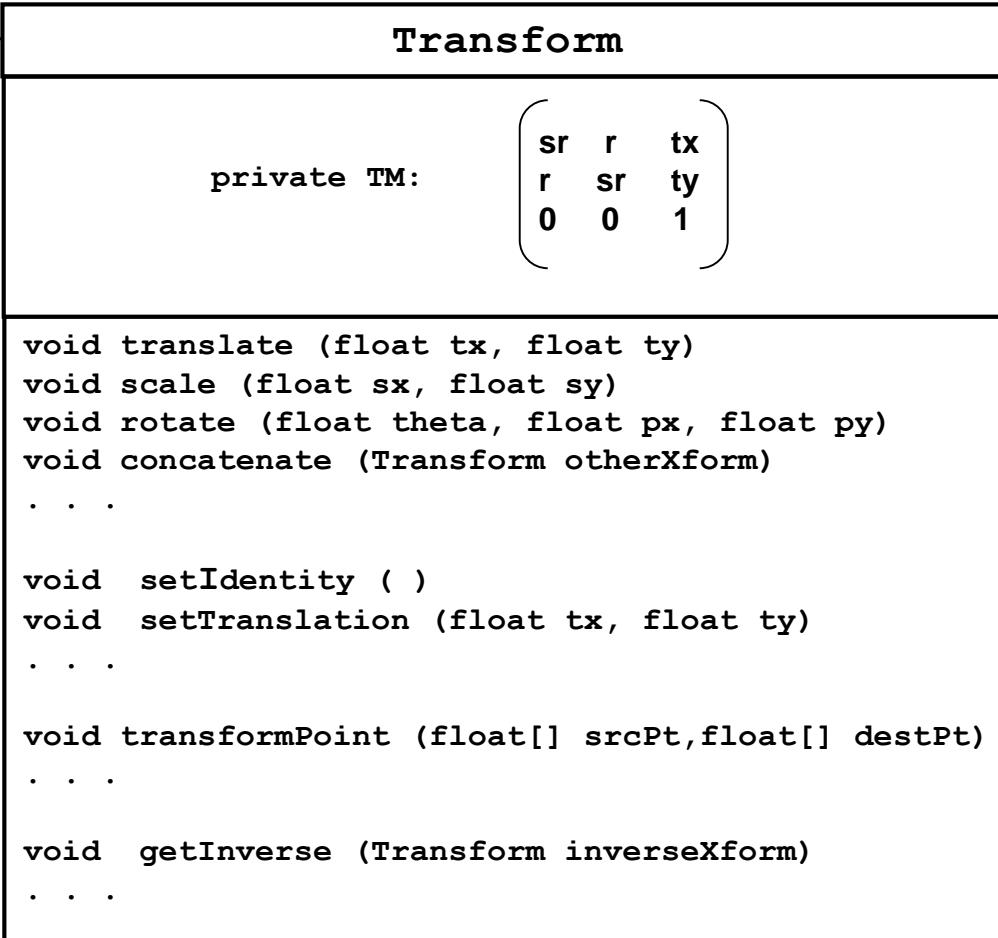
Methods to *apply* the transform (xform) to other objects

- To initialize use the following static function:

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

Transform Objects

myXform



} Modify TM

} Replace TM

} Apply xform to other objects

} Utilities

- Methods for modifying 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};
// Initialize
Transform myXform = Transform.makeIdentity();
myXform.rotate(Math.toRadians(45), 0, 0);
myXform.transformPoint (p1,p2);
```

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

“Local” Coordinate Systems

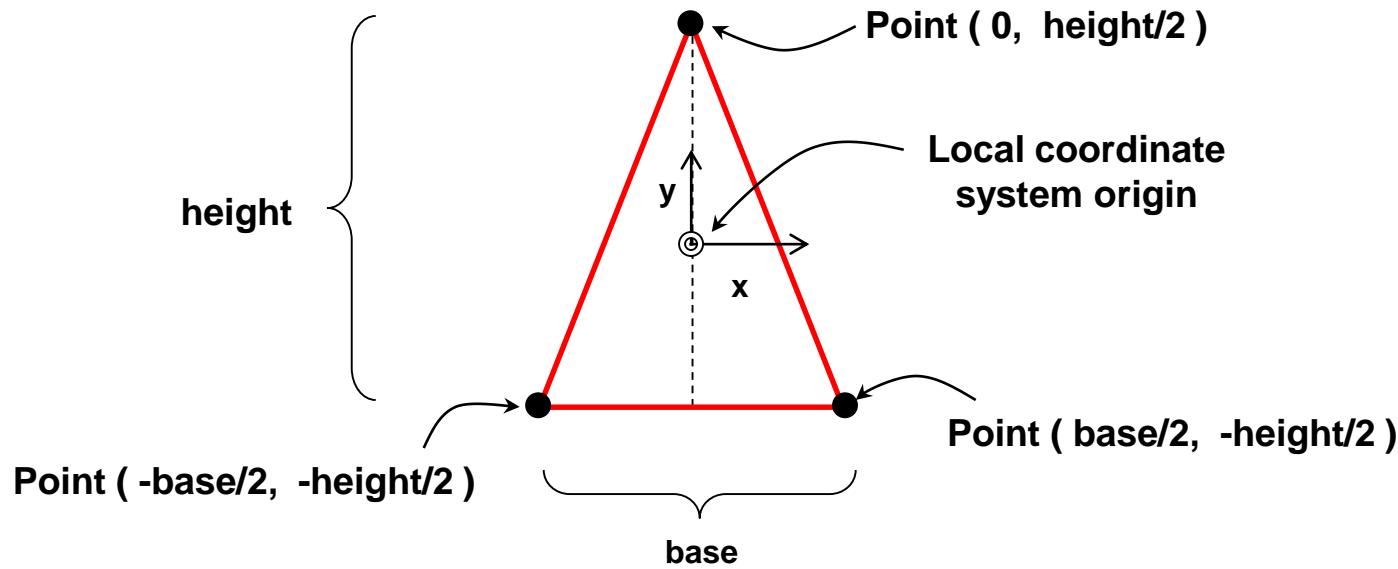
- It is the coordinate system the model was made in.
- A simple example is using house numbers to locate a house on a street; the street is a **local coordinate system** within a larger **system** composed of city townships, states, countries, etc.

Source: https://en.wikipedia.org/wiki/Local_coordinates

“Local” Coordinate Systems

Define objects *relative to their own origin*

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



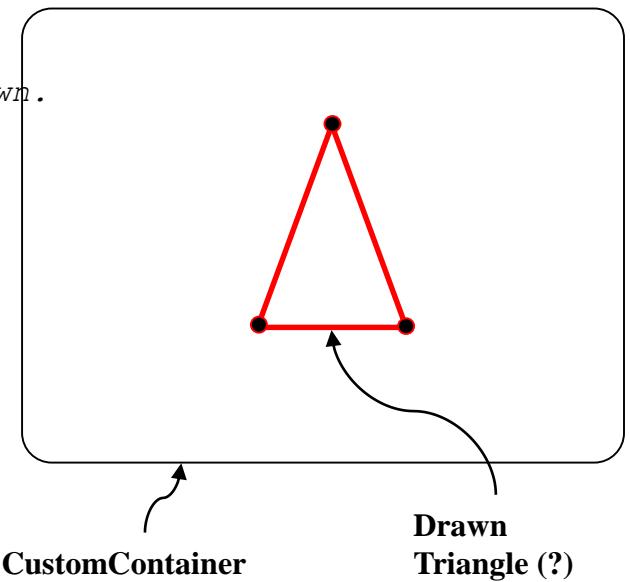
Triangle Class

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

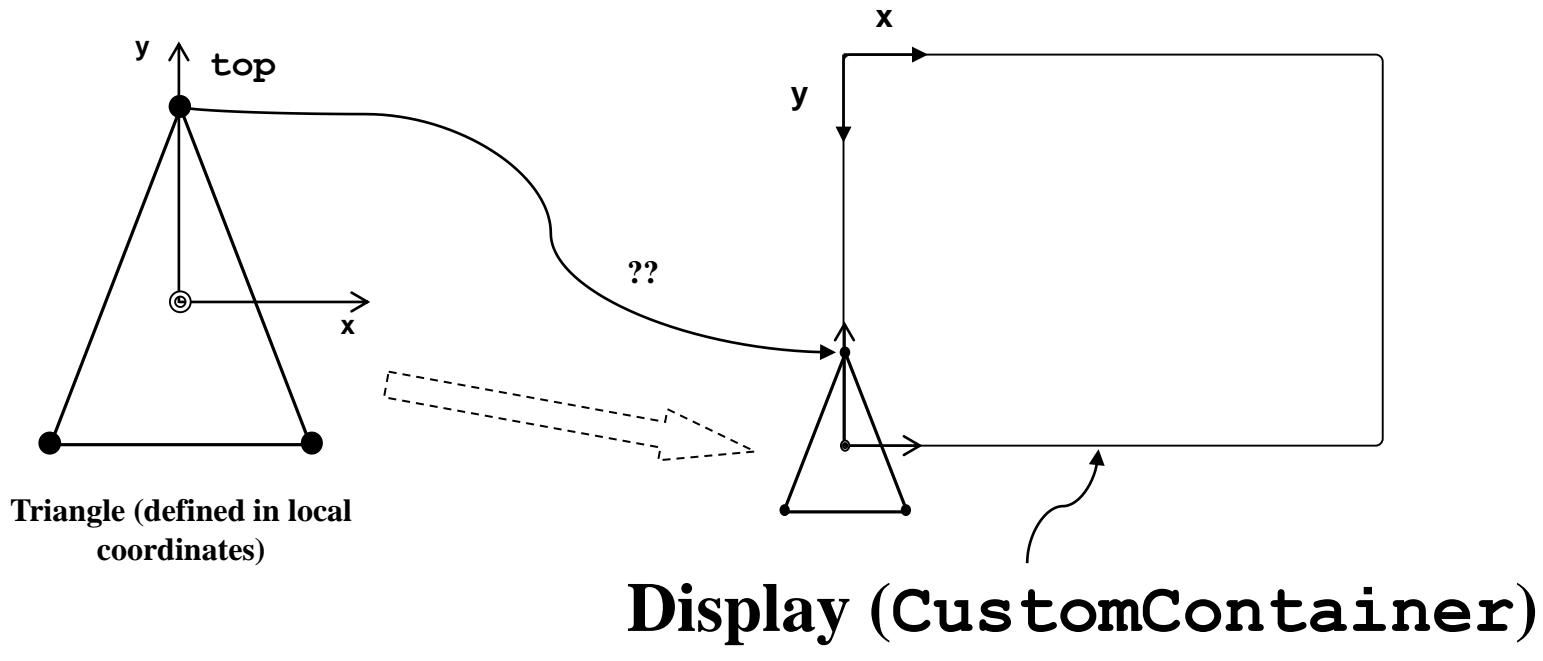
Drawing A Triangle

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



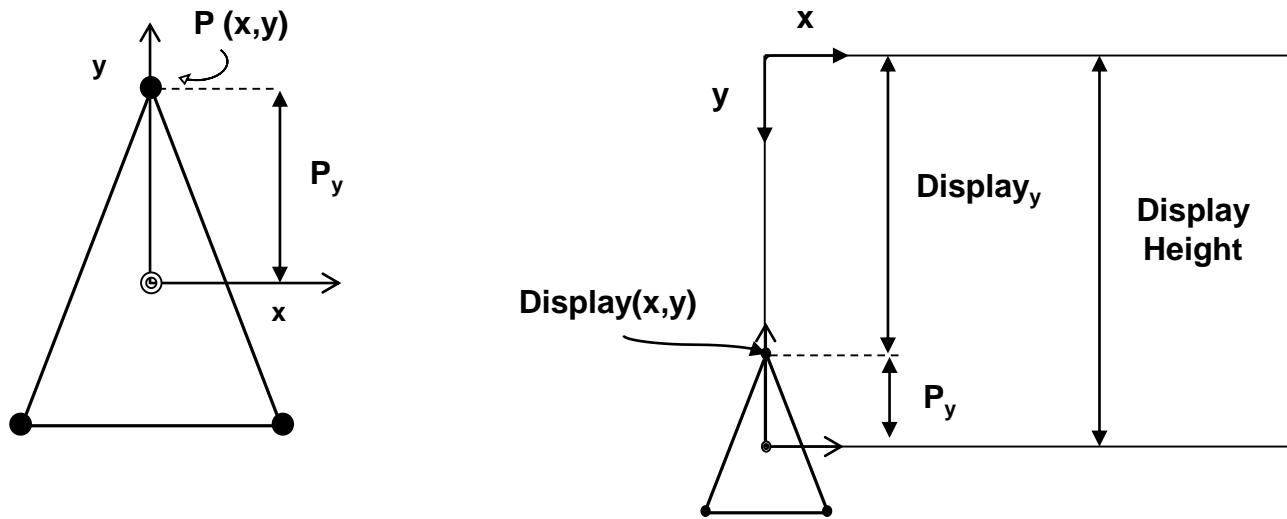
Mapping To Display Location

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



Mapping To Display Location

(cont.)



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

$$= \underbrace{(-1 * (P_y))}_{\text{Scale}_y(-1)} + \underbrace{\text{DisplayHeight}}_{\text{Translate}_y(\text{DisplayHeight})}$$

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);

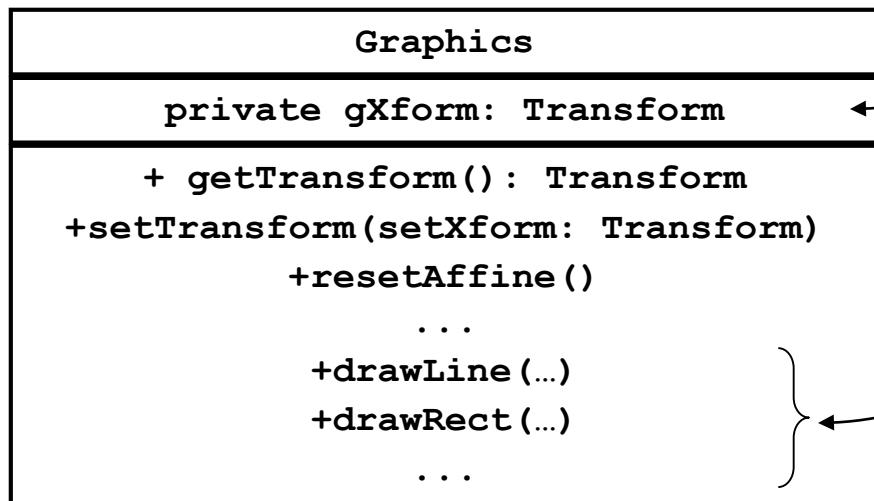
        // draw the (transformed) triangle
        g.setColor(color);
        g.drawLine(pCmpRelPrnt.getX()+(int)top[0], pCmpRelPrnt.getY()+(int)top[1],
                  pCmpRelPrnt.getX()+(int)bottomLeft[0],
                  pCmpRelPrnt.getY()+(int)bottomLeft[1]); // left side
        g.drawLine(pCmpRelPrnt.getX()+(int)bottomLeft[0],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

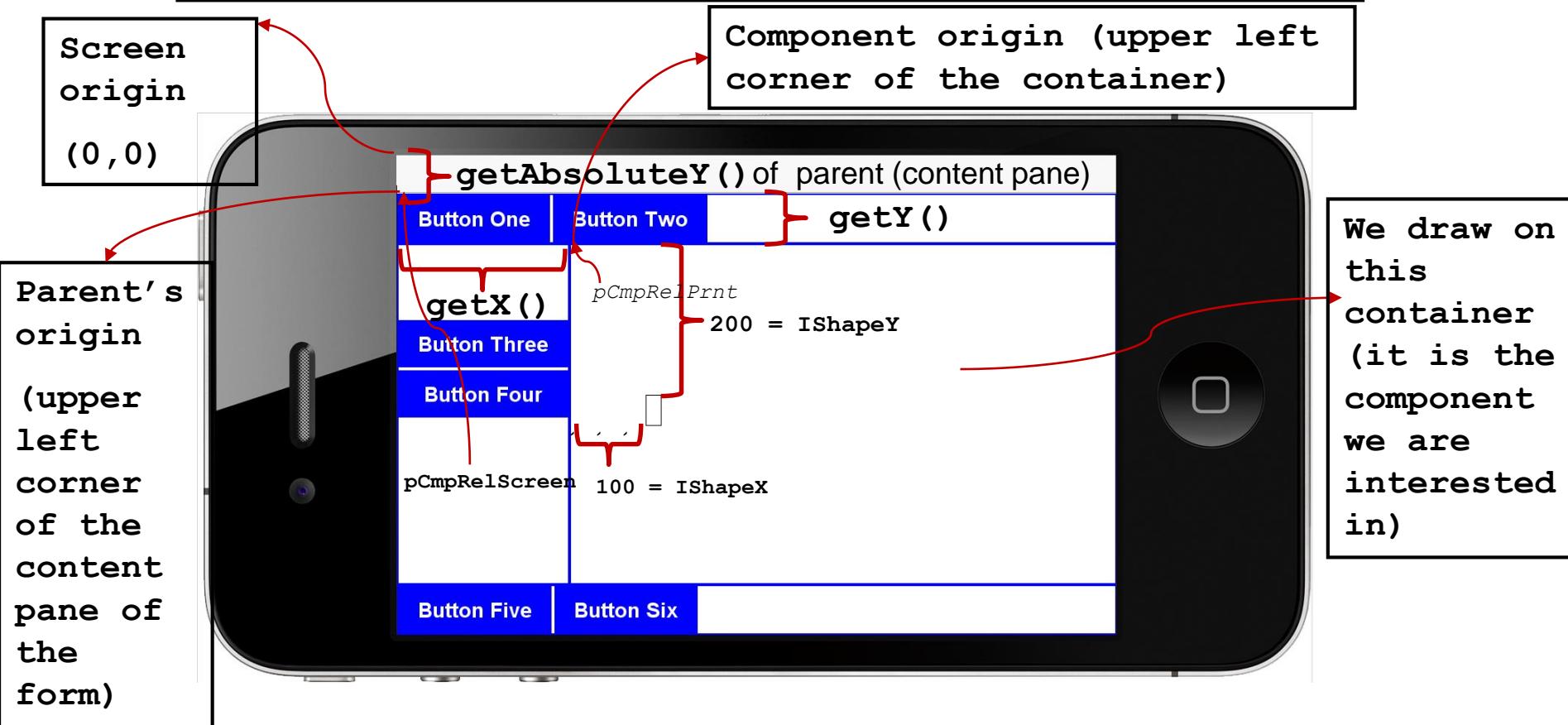
- Every **Graphics** contains a **Transform** object
 - This transform is applied to all drawing coordinates during drawing



gXform has the current xform of the Graphics object

All drawing methods apply current xform to drawing coordinates

pCmpRelScreen vs pCmpRelPrnt



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)
```

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.
- 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.)

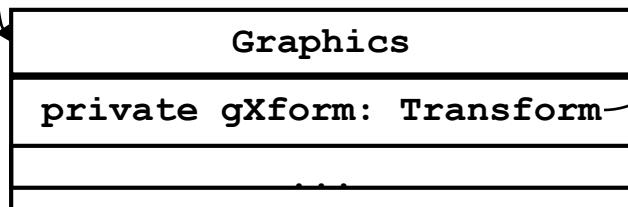
```
public class CustomContainer extends Container {  
    private Triangle myTriangle ;  
    public CustomContainer () {  
        myTriangle = new Triangle (200, 200);  
    }  
  
    public void paint (Graphics g) {  
        super.paint(g);  
        Transform gXform = Transform.makeIdentity();  
        g.getTransform(gXform);  
        //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())
```

g



`gXform` has the current xform of the `Graphics` object `g`.

Initially transformation matrix (TM) in `gXform` is not identity. It has a previous content `[M]`. After transformations are applied to `gXform`, TM would be equal to this:

$$\underbrace{[M]}_{\text{Previous content}} \times [T(\text{abs}X, \text{abs}Y)] \times [T_y(\text{displayHeight})] \times [S_y(-1)] \times [T(-\text{abs}X, -\text{abs}Y)]$$

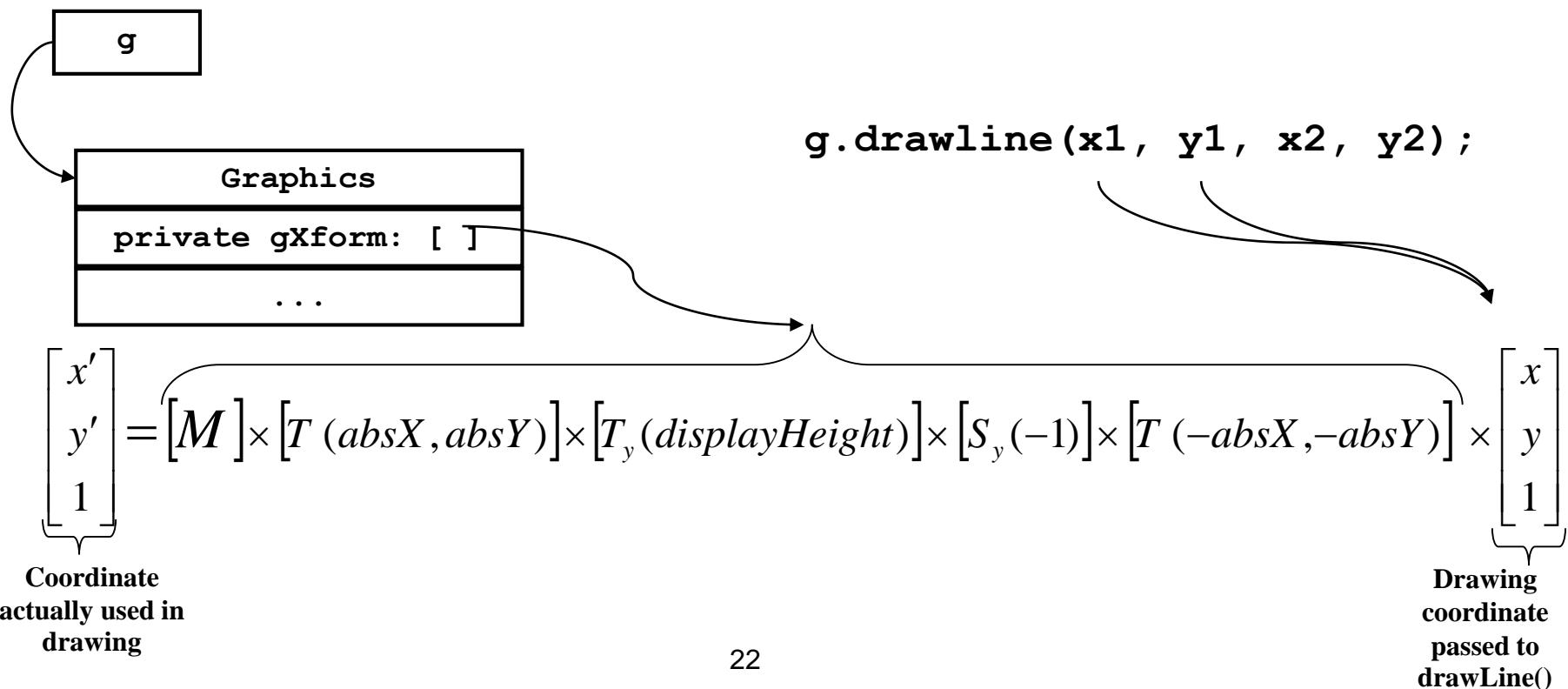
Previous content

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()) ;  
    }  
}
```

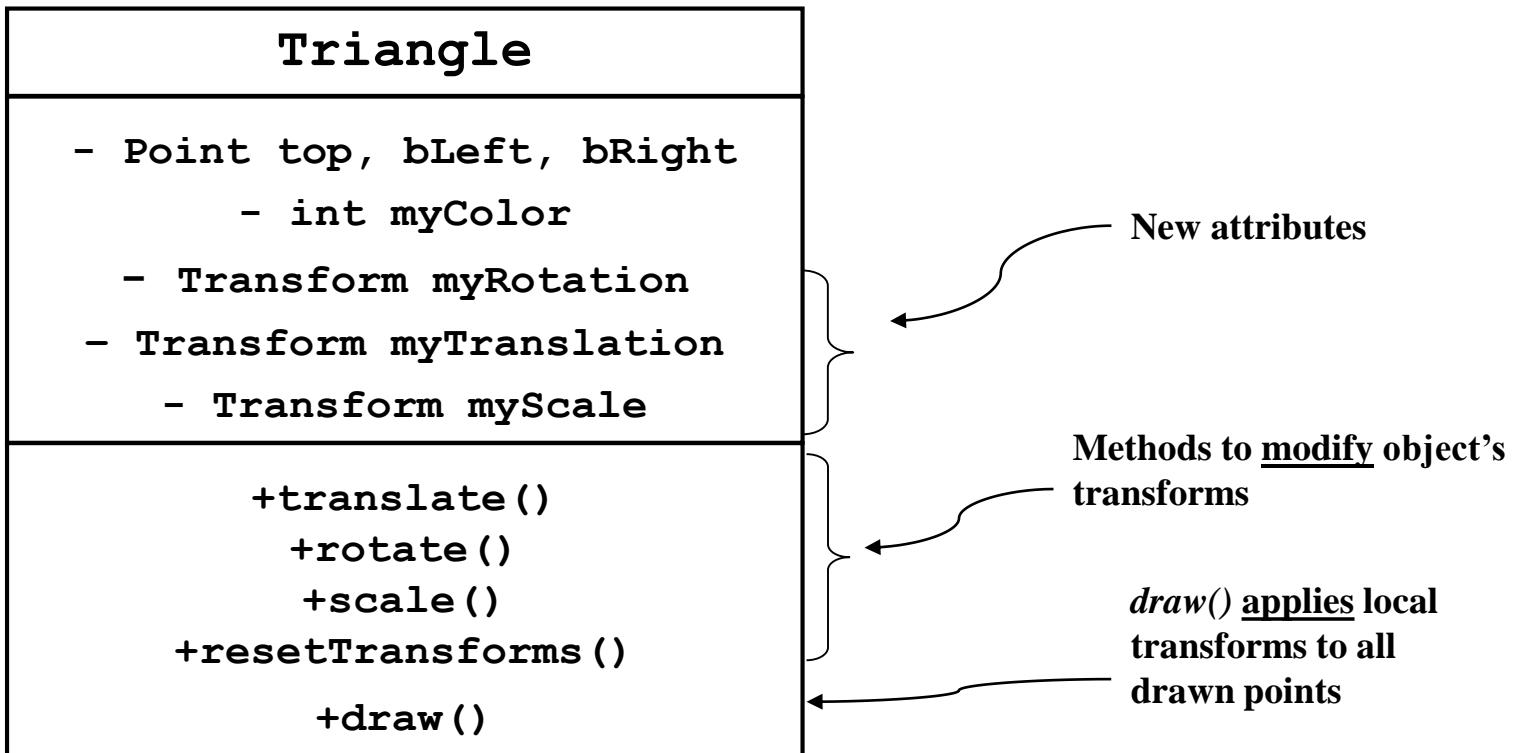
Using Graphics's Xform (cont.)

- 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.makeIdentity();  
    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

Order of application of transformations

drawing coordinate passed to drawLine()

Also 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 (want) for the interactive sequence “Drag₁, Click₁, Drag₂, Click₂” 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:

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

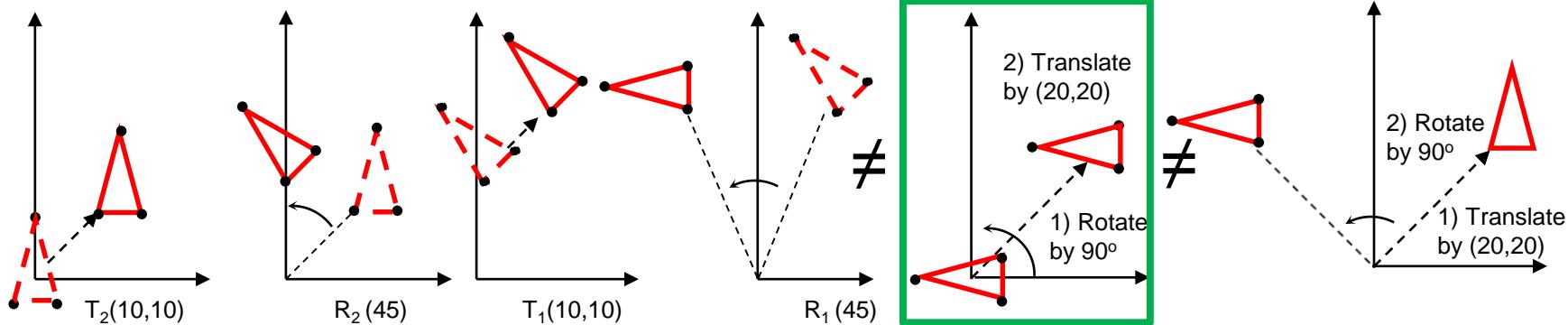
(by default xform is an identity matrix and it is modified by multiplications **on the right**)

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:

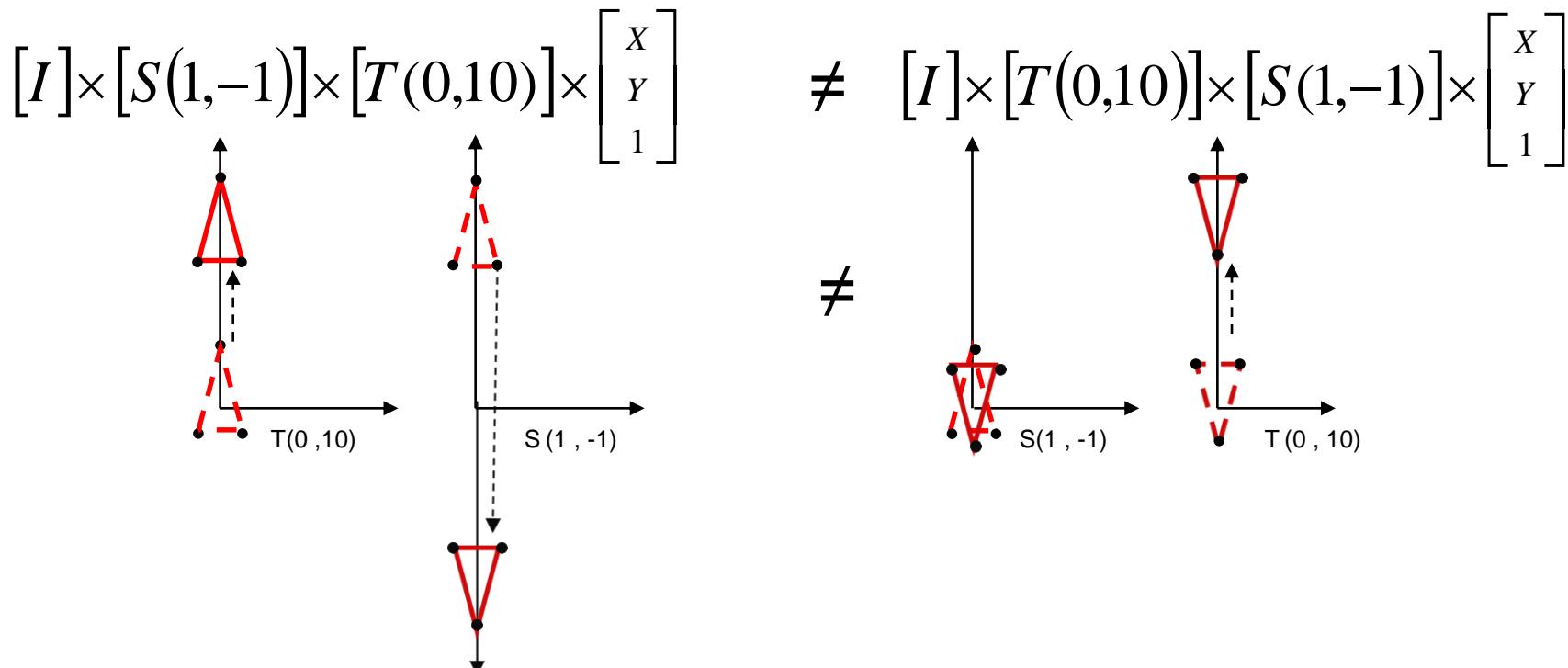
$$[I] \times [R_1(45)] \times [T_1(10,10)] \times [R_2(45)] \times [T_2(10,10)] \times \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

(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...

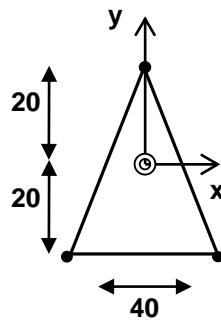
On Transform Order and Number of LTs (cont.)



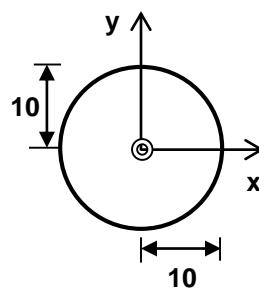
If Click = translate (0,10), Drag = scale (1, 2) and the expected result for “Drag₁, Click₁, Drag₂, Click₂” is: “Scaling the height of triangle by x4, and Translation by a total of (0,20)” Then we should have a separate transform for accumulating scaling transformations too... (Then we would use these 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

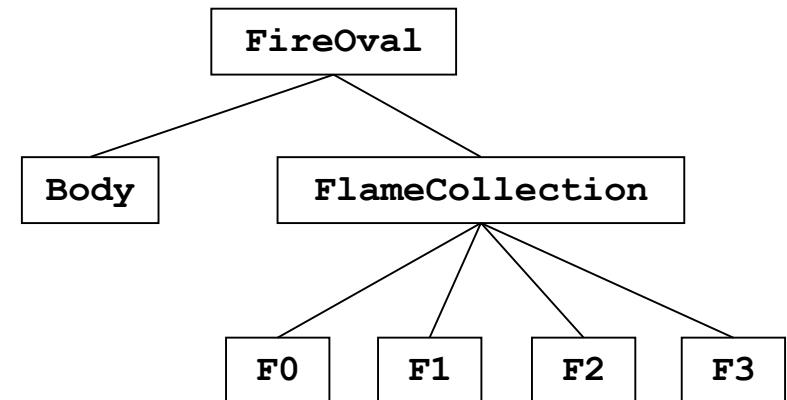
- We can build an object by combining
 - Simpler “parts”
 - Transformations to “orient” the parts



A “Flame” object



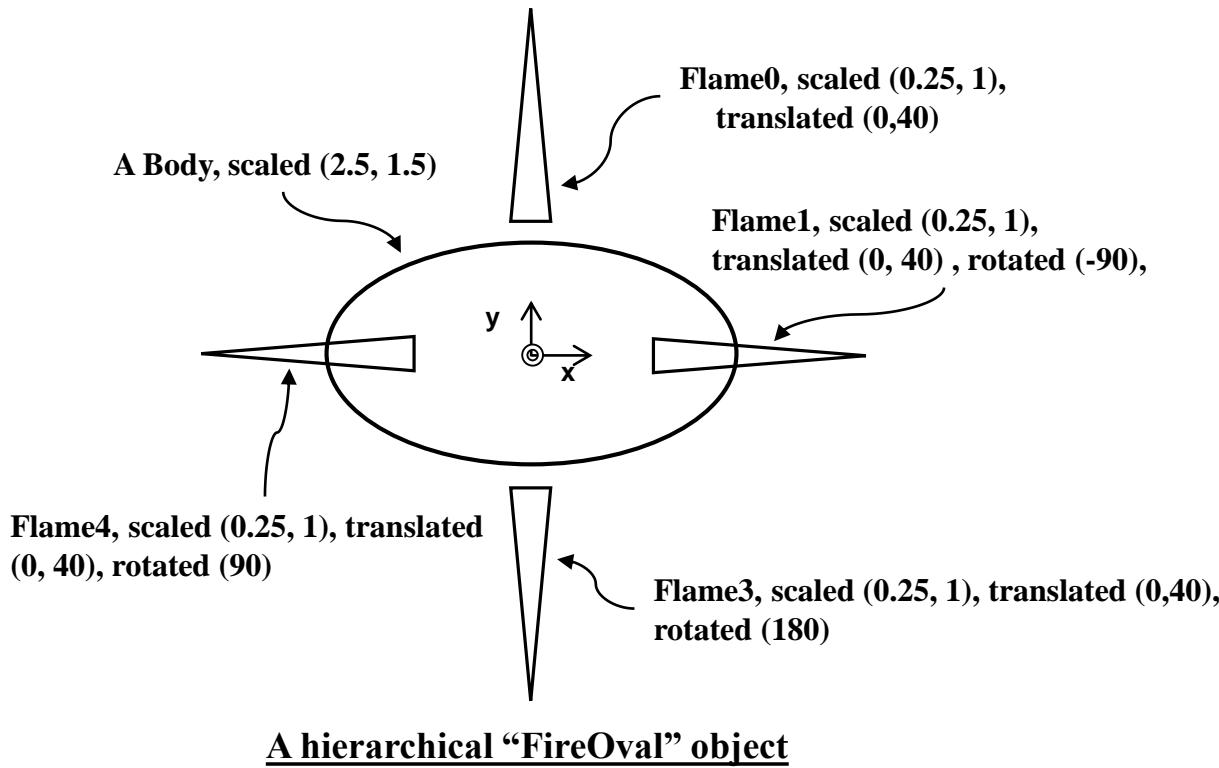
A “Body” object



A hierarchical “FireOval” object

Hierarchical Objects (cont.)

- FireOval Transformations



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 flame'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.makeIdentity();
    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...
```

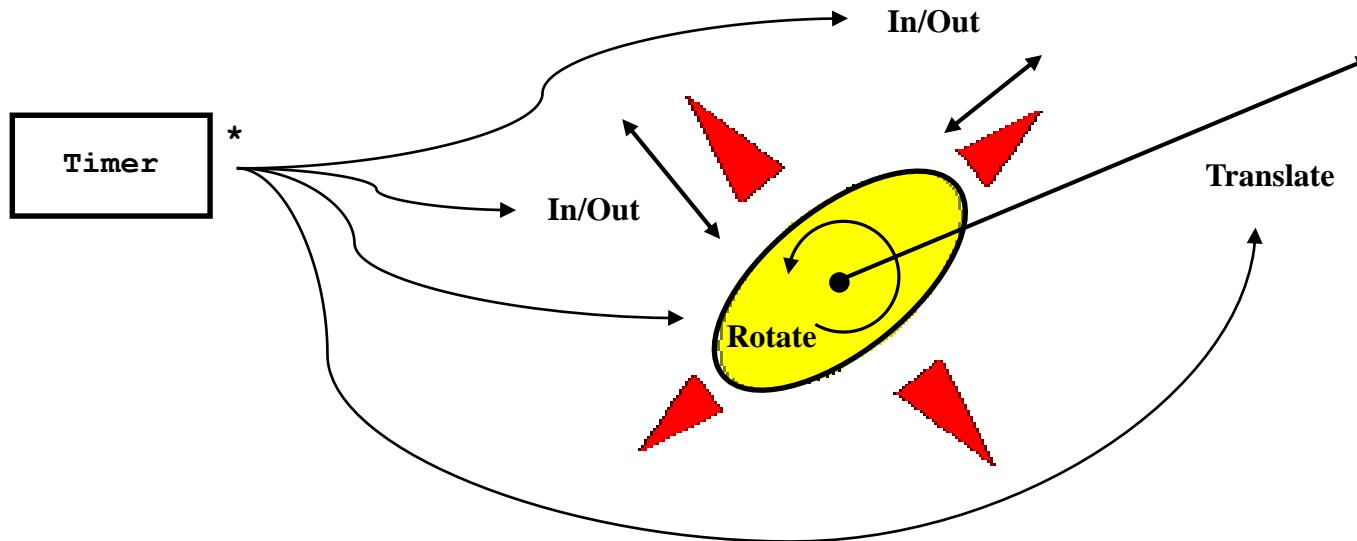
Hierarchical Objects (cont.)

```
// 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 and 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.makeTextform();
        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()! CSc Dept, Sac State
          //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.)

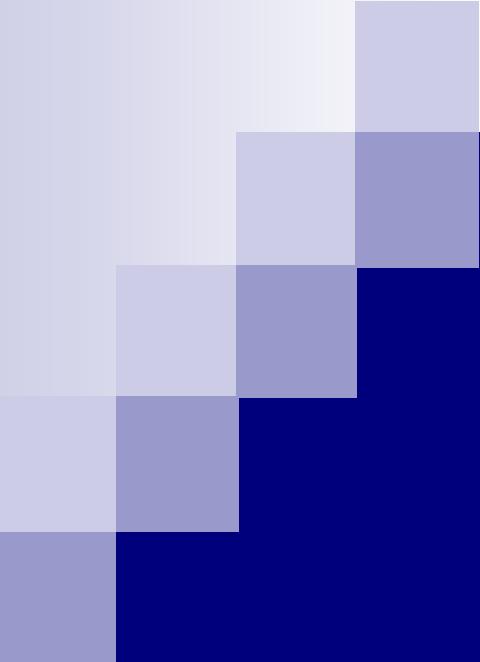
```
/** This class defines a Form containing a CustomContainer object that displays
 * the FireOval. It uses a Timer to call updateLTs() which modify FireOval's and
 * its Flames' local transformations.
 * CustomContainer class looks exactly like the one used in static FireOval
 * example expect it also 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 ;
        }
    }
}
```



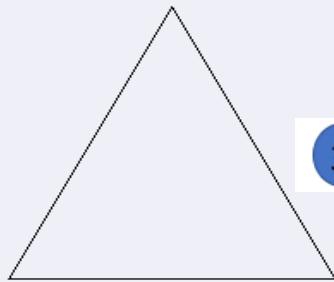
15 - Viewing Transformations

Computer Science Department
California State University, Sacramento

Recall from Module 14 (Applications of Affine Transforms)

1

Demo 1: A triangle is partially displayed where we have its local origin coincided with its container origin. It is drawn up side down.



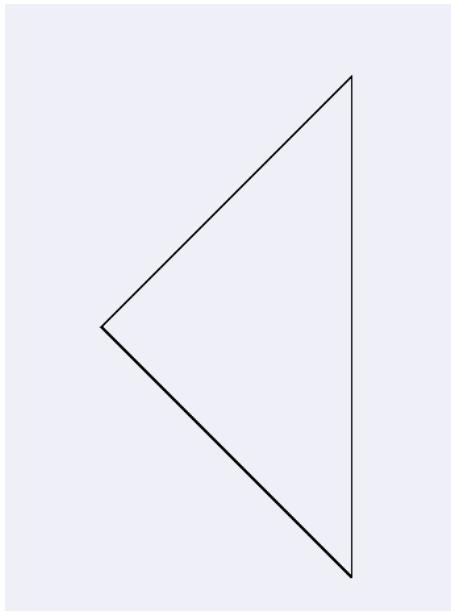
3

Demo 3: When applying local transformations (LTs) and display mapping transformations, the full triangle is drawn in a display. (not mention local origins transformation)

2

Demo 2: Modifying Graphics's Xform , when applying display mapping (scaling and translation) and local origin transformations, the triangle is drawn right side up.

Module 14 (Applications of Affine Transforms Implementation cont)



Local Transformations (LTs): Local transformations applied to a specific item to be drawn.

```
//apply some transformations to the triangle
myTriangle.translate (300, 300);
myTriangle.rotate (90);
myTriangle.scale (2, 1);
```



Display Mapping Transform: A transformation applies to objects to present them as display - in a right side up manner!

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

Local Origin Transformation: A transform to make local origin coincides with screen origin before the display mapping transformation (or other transform) is done.

```
draw(Graphics g, Point
      pCmpRelPrnt, point pCmpRelScrn)
{ save xform in g as gOrigXform;
  apply LTs and "local origin"
        transforms to g;
  g.drawLine...
  restore xform in g with
        setTransform(gOrigXform);
}
```

Recall: 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

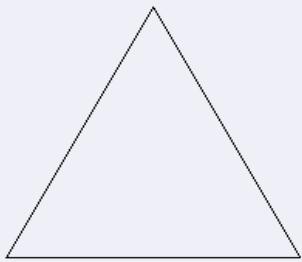
Order of application of transformations

drawing coordinate passed to drawLine()

Also 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.

Panning & zooming of a Triangle



pan right: The camera swivels to the right, causing the image to move from right to left across the screen (image would shift left.)

pan left: The camera swivels to the left, causing the image to move from left to right across the screen (image would shift right.)

Zooming Out: two fingers come closer together. Make image becomes smaller.

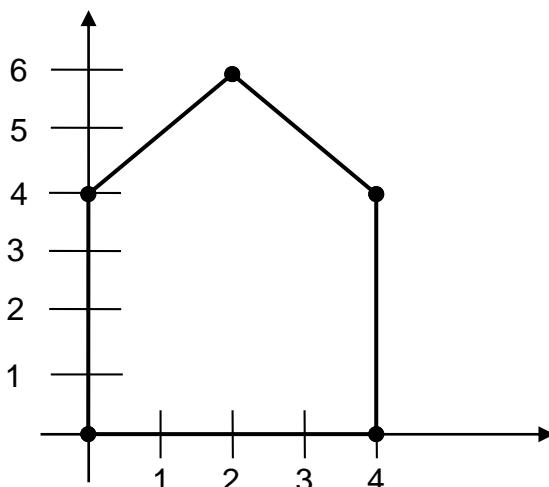
Zooming In: two fingers go away from each other. Make image becomes bigger.

Overview

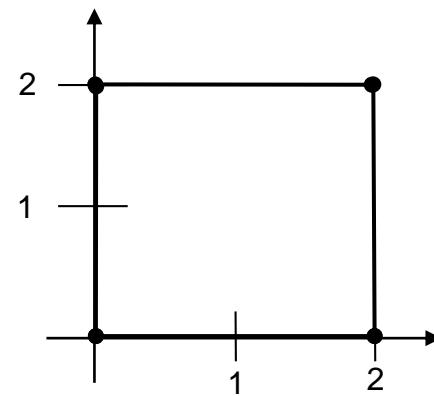
- **The World Coordinate System**
- **Mapping From World to Display Coordinates**
 - **World Window, Normalized Device (ND), World-to-ND Transform, ND-to-Display Transform, the Viewing Transformation Matrix (VTM)**
- **2D Viewing Operations (Zoom and Pan)**
- **Clipping and the Cohen-Sutherland Algorithm**

Local Coordinate Systems

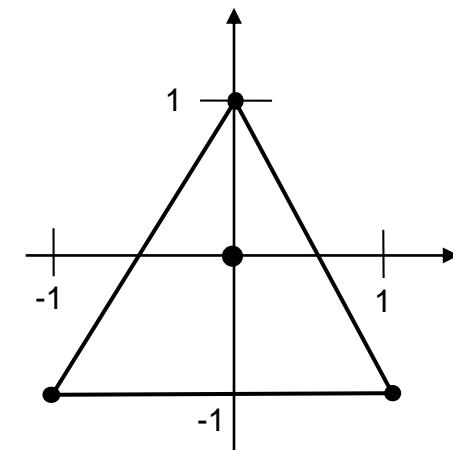
- Each object is defined in its “own space”
- Each object has its local origin within a local coordinates



Pentagon

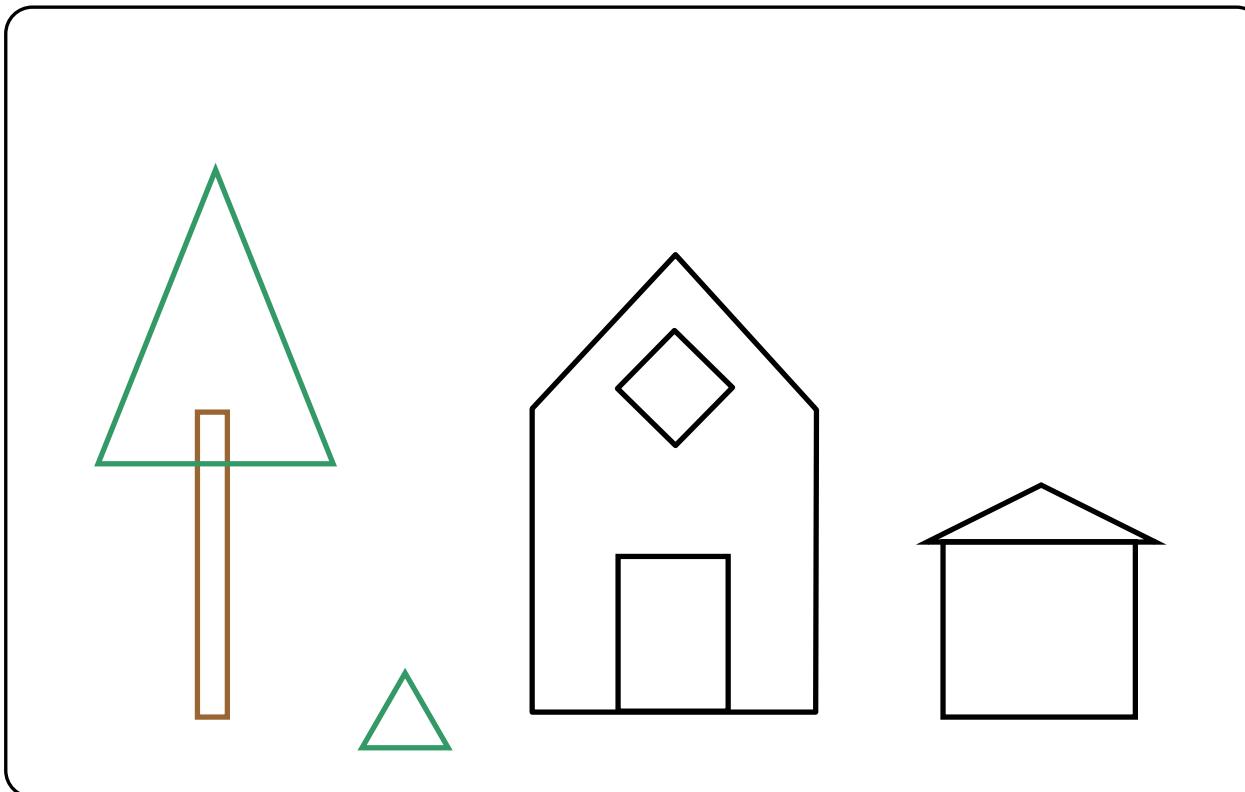


Box



Triangle

Creating A “World”



Tree
(Triangle + Box)

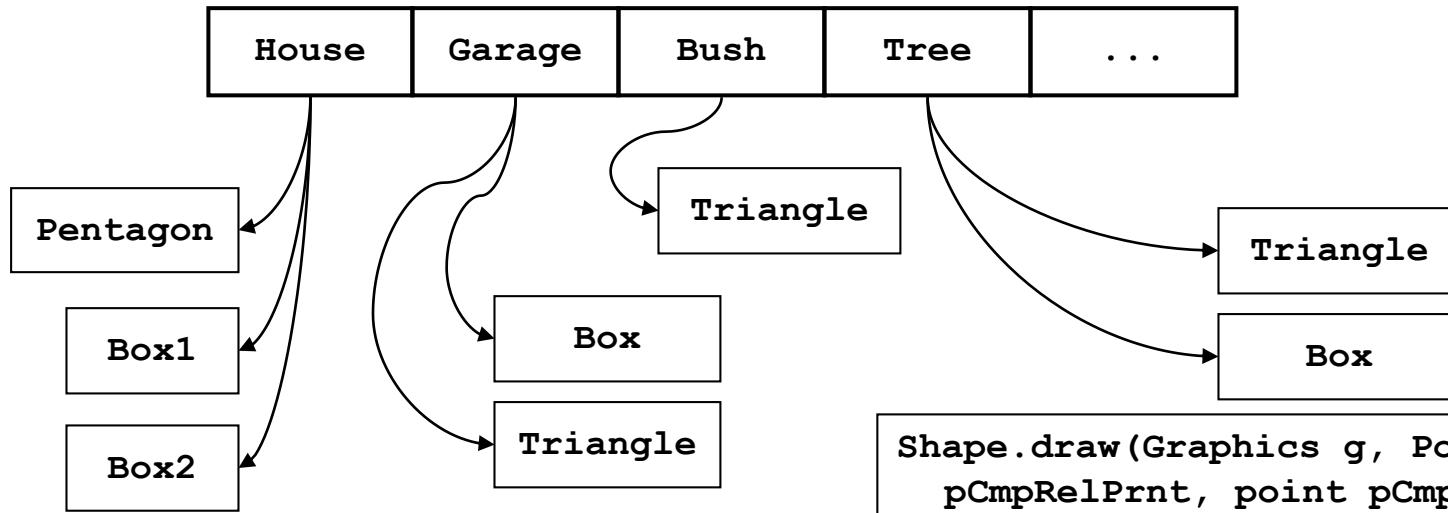
Bush
(Triangle)

House
(Pentagon + Two Boxes)

Garage
(Box + Triangle)

The World Object Collection

`worldShapeCollection`



```

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);

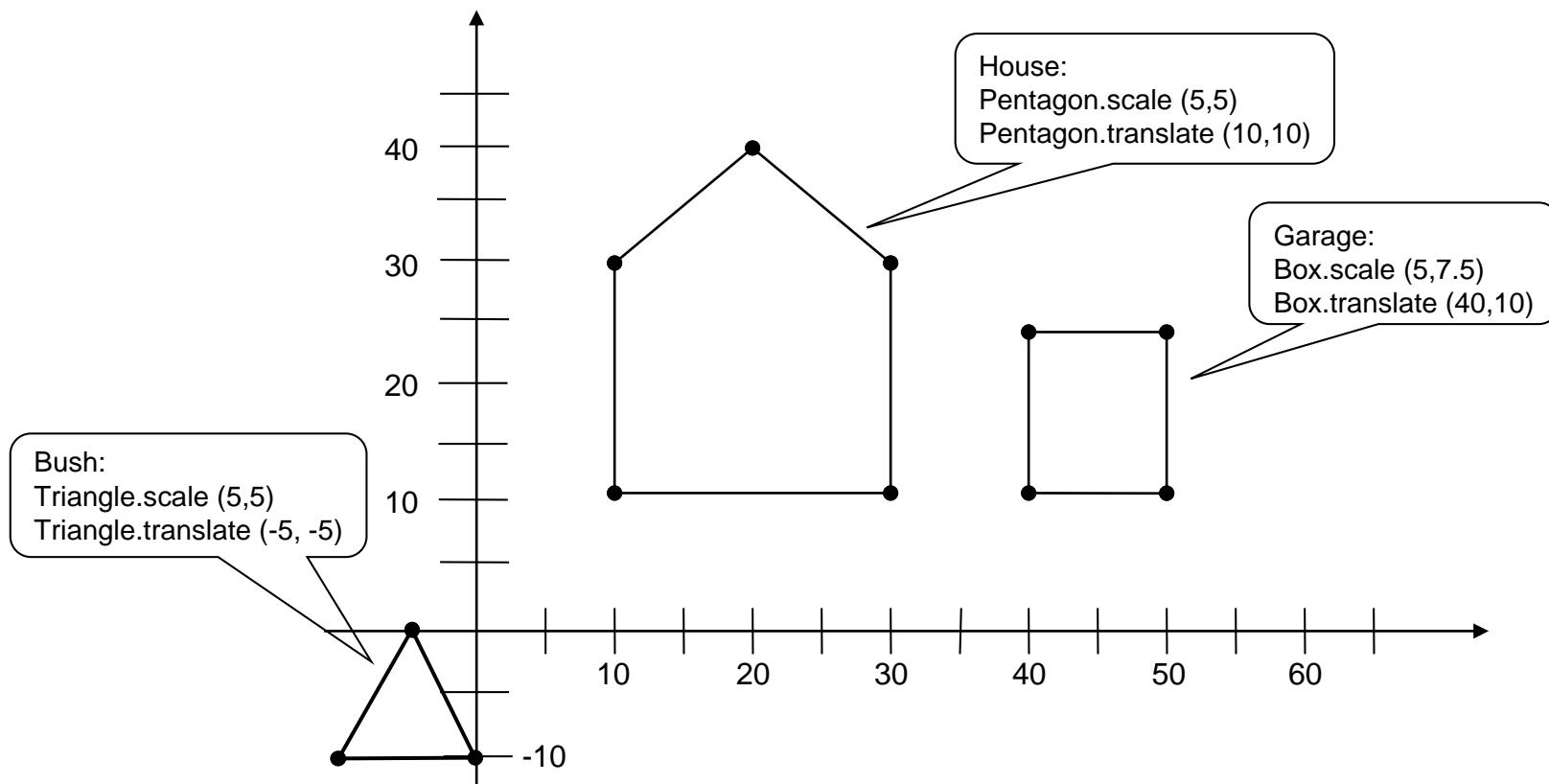
  }
  restore xform in g with
  setTransform(gOrigXfrom);
}
  
```

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:



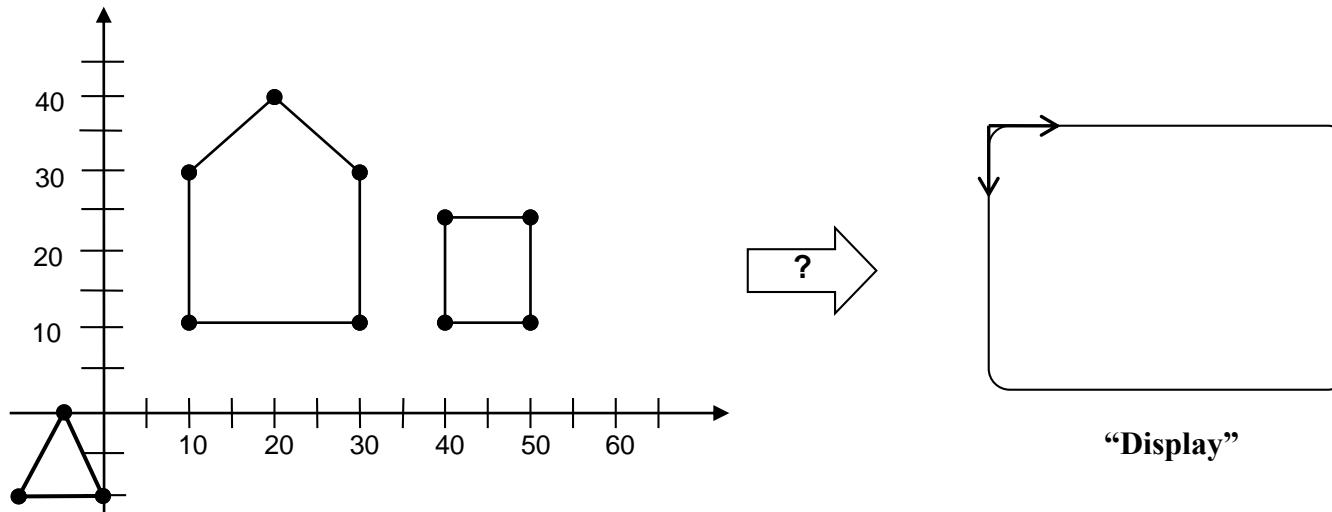
Local Transformations

- 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:

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



Drawing The World (cont.)

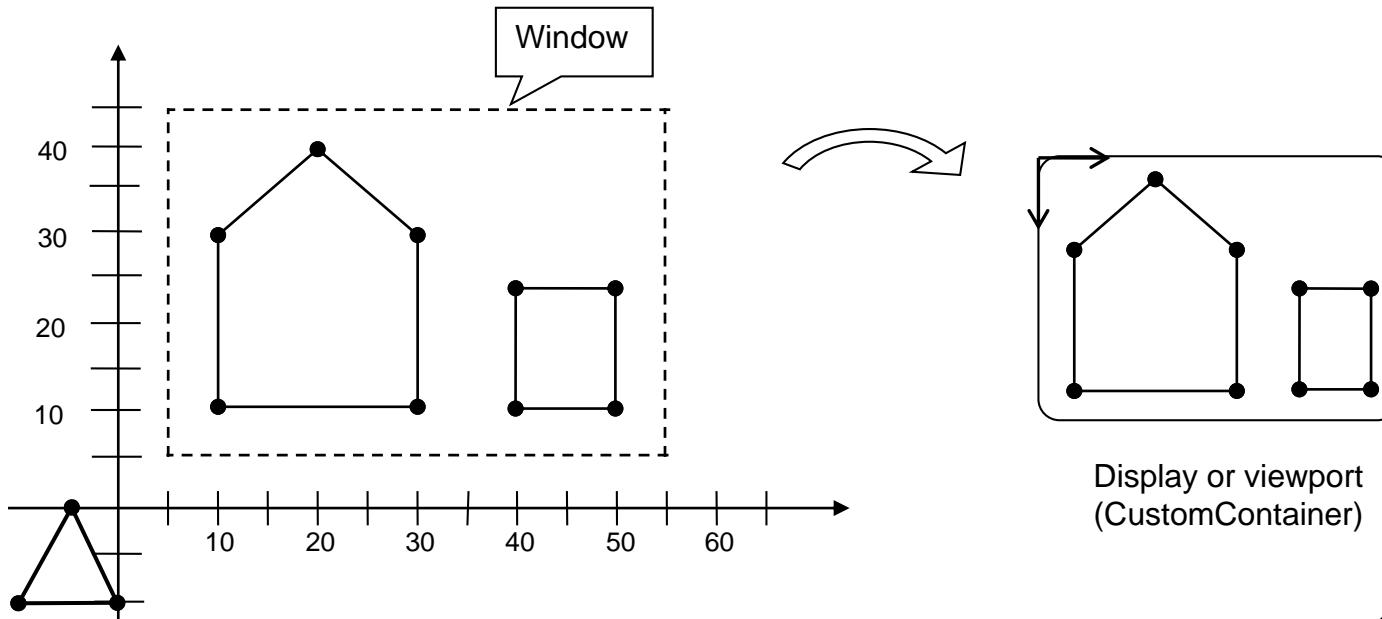
Solution:

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

The World “Window”

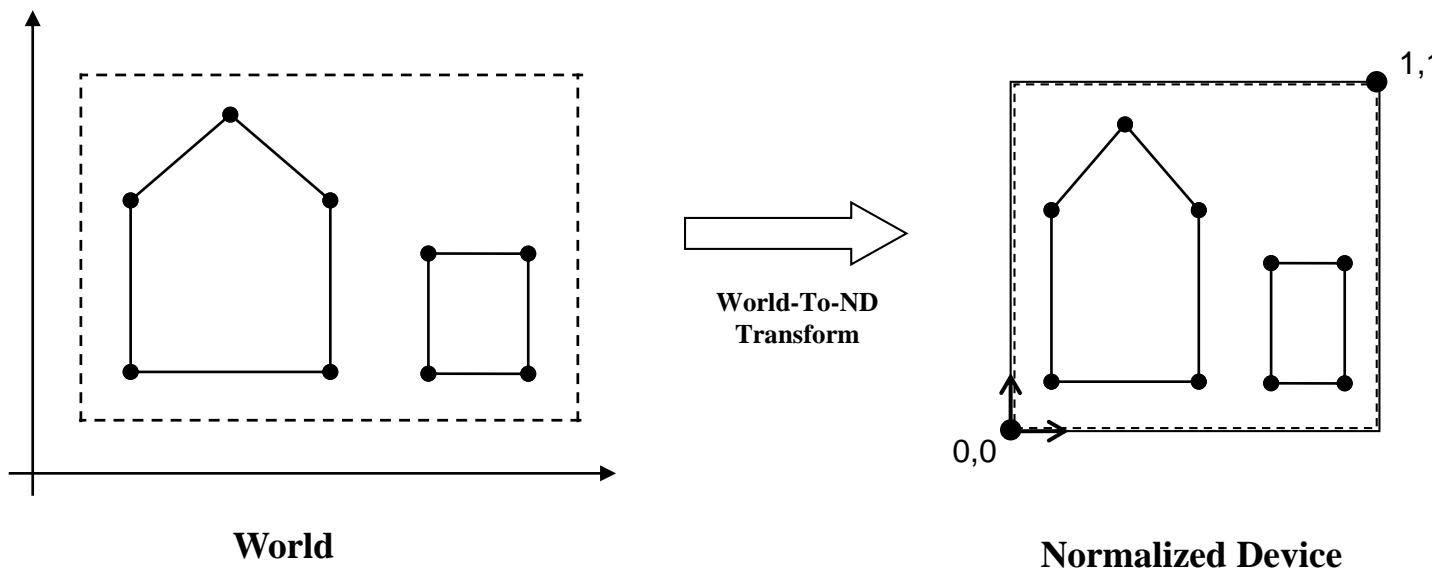
Defines the part of the world that appears on display

- 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”

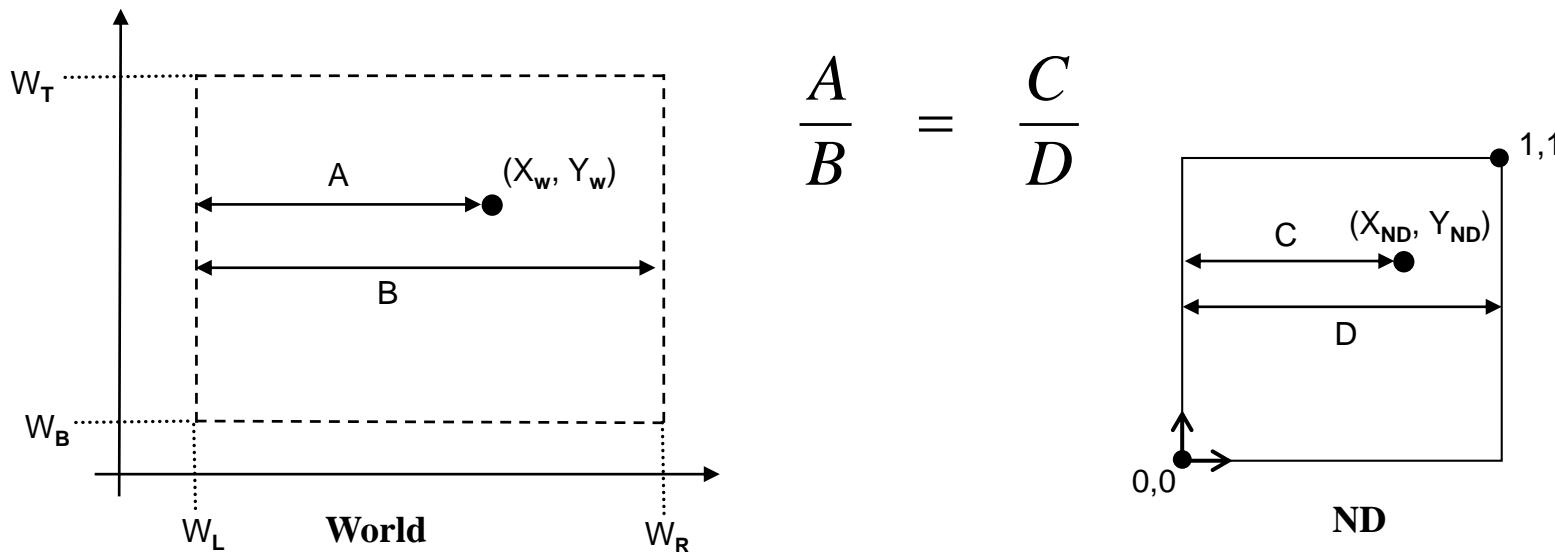
- Properties of the Normalized Device (ND):
 - Square
 - Fractional Coordinates (0.0 .. 1.0)
 - Origin at Lower Left
 - Corners correspond to world window



World-To-ND Transform

Consider a single point's X coordinate

- Need to achieve proportional positioning on the ND



$$A = X_w - W_L ; \quad B = W_R - W_L ; \quad D = 1 ;$$

$$\therefore C = X_{ND} = \frac{(X_w - W_L)}{(W_R - W_L)} = (X_w - W_L) * \frac{1}{(W_R - W_L)}$$

World-To-ND Transform (cont.)

Consider the form of X_{ND} :

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

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

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

World-To-ND Transform (cont.)

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

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

or

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

- In Matrix Form:

$$\begin{pmatrix} X_{\text{ND}} \\ Y_{\text{ND}} \\ 1 \end{pmatrix} = \begin{pmatrix} 1/w_w & 0 & 0 \\ 0 & 1/w_h & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -W_L \\ 0 & 1 & -W_B \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_w \\ Y_w \\ 1 \end{pmatrix}$$

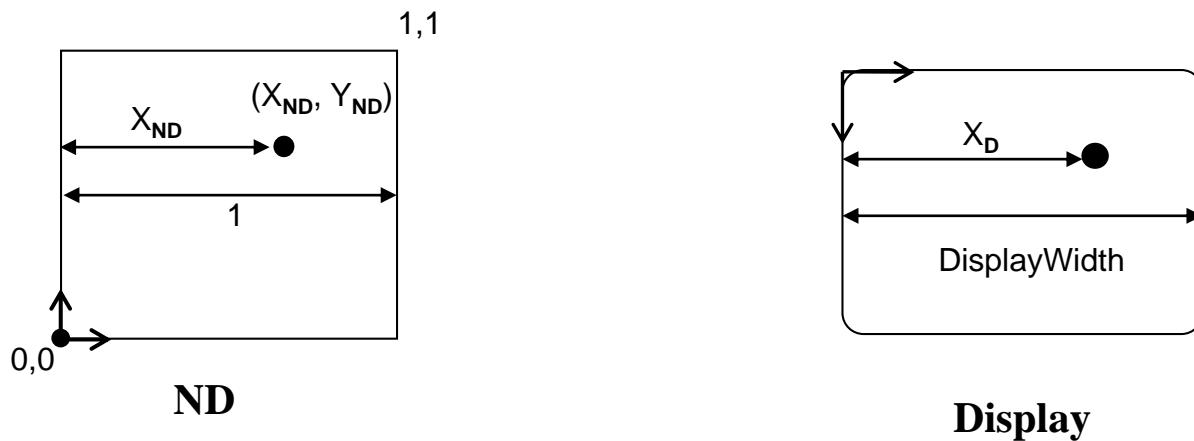
X

World-to-Normalized-Device (W2ND) Transform

$$\begin{pmatrix} X_{\text{ND}} \\ Y_{\text{ND}} \\ 1 \end{pmatrix} = \begin{pmatrix} X_w \\ Y_w \\ 1 \end{pmatrix}$$

ND-To-Display Transform

- A similar approach can be applied

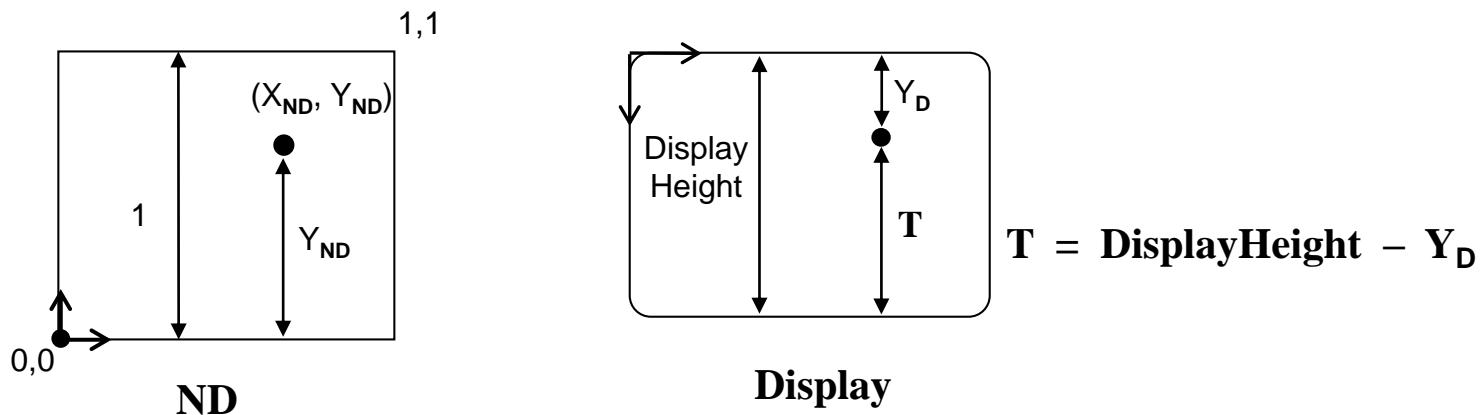


$$\frac{X_{ND}}{1} = \frac{X_D}{DisplayWidth} ; \therefore X_D = X_{ND} \times DisplayWidth$$

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

ND-To-Display Transform (cont.)

- Similarly for height:



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

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

ND-To-Display Transform (cont.)

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

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

or

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

- In Matrix Form:

$$\begin{bmatrix} X_D \\ Y_D \\ 1 \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & D_{height} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} D_{width} & 0 & 0 \\ 0 & -D_{height} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} X_{ND} \\ Y_{ND} \\ 1 \end{bmatrix}$$


ND-to-Display Transform

Combining Transforms

$$\begin{pmatrix} x_D \\ y_D \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} \text{ND} \\ \text{to} \\ \text{Display} \end{pmatrix} \times \left(\begin{pmatrix} \text{World} \\ \text{to} \\ \text{ND} \end{pmatrix} \times \begin{pmatrix} x_W \\ y_W \\ 1 \end{pmatrix} \right)}_{\text{VTM}}$$

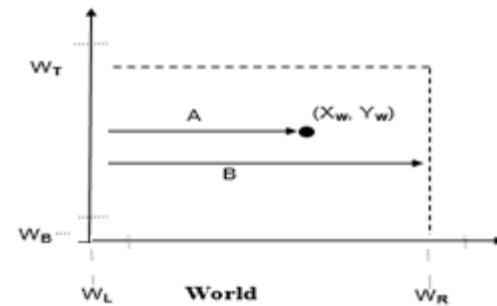
$$\begin{pmatrix} x_D \\ y_D \\ 1 \end{pmatrix} = \begin{pmatrix} \text{“VTM”} \end{pmatrix} \begin{pmatrix} x_W \\ y_W \\ 1 \end{pmatrix}$$

Using The VTM

- Suppose we have
 - A 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 ...  
    }
```



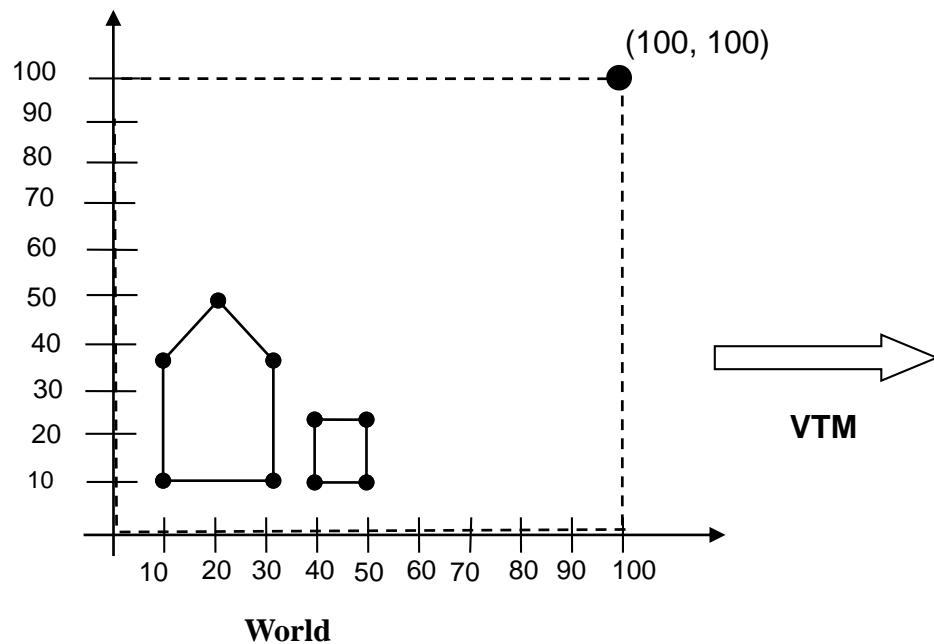
Using The VTM (cont.)

... continued ...

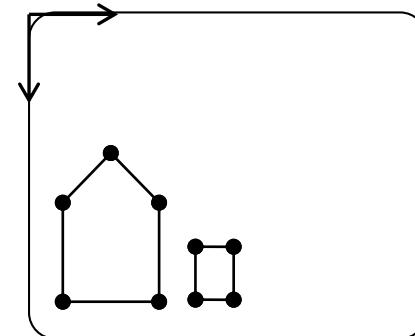
```
// concatenate the VTM onto the g's current transformation (do not forget to apply "local
//origin" transformation)
Transform gXform = Transform.makeIdentity();
g.getTransform(gXfrom);
gXform.translate(getAbsoluteX(),getAbsoluteY()); //local origin xform (part 2)
gXform.concatenate(theVTM); //VTM 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:



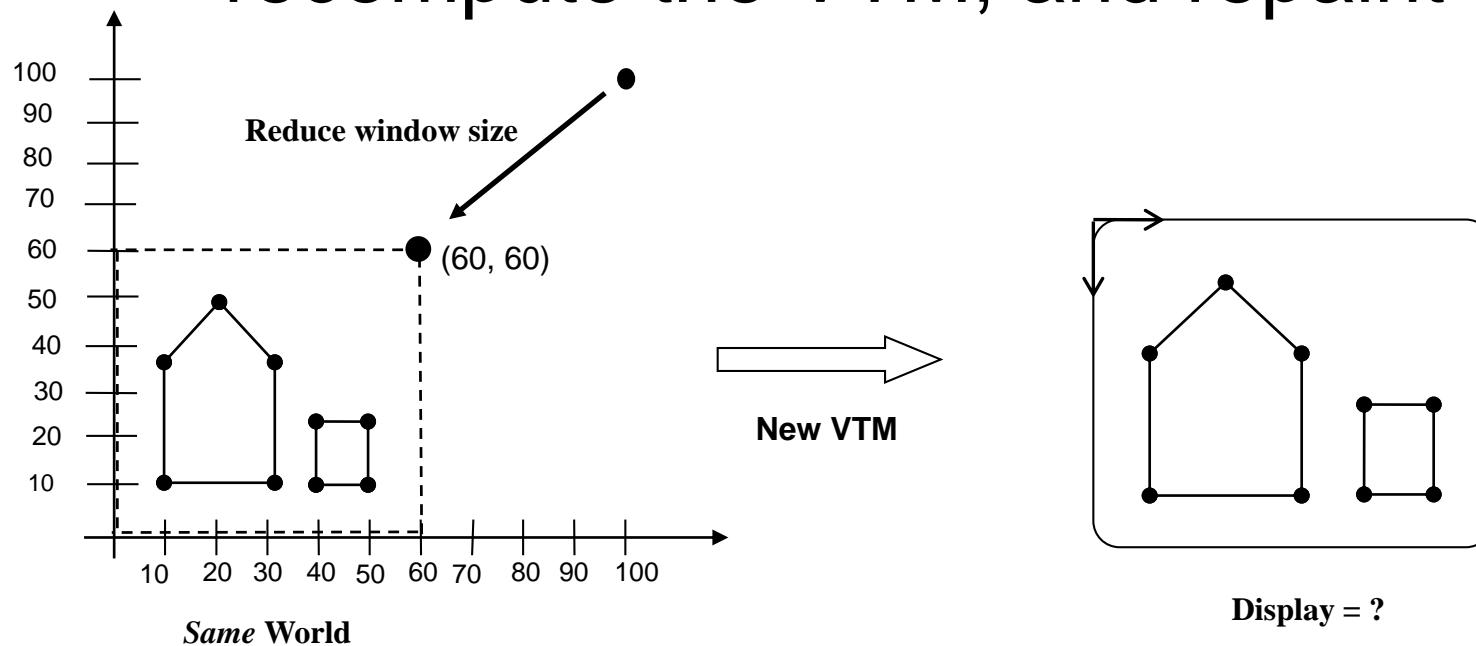
VTM



Display

Changing the Window Size (cont.)

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

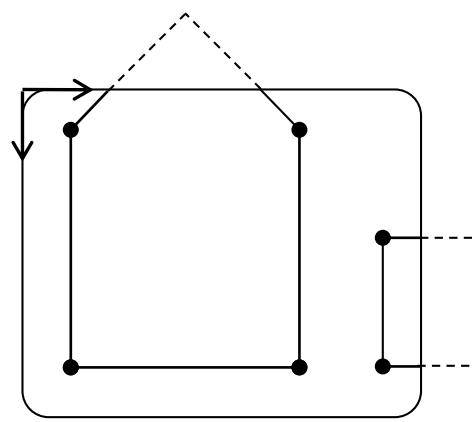
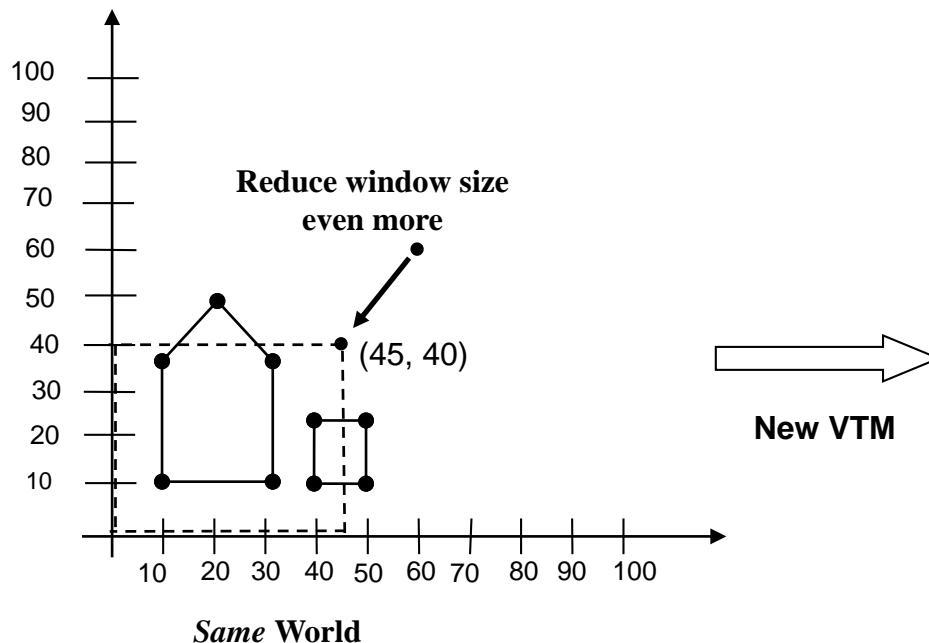


$$X_{ND} = (X_w - W_L) * \frac{1}{(W_R - W_L)}$$

$$X_D = X_{ND} * Scale(DisplayWidth)$$

Changing the Window Size (cont.)

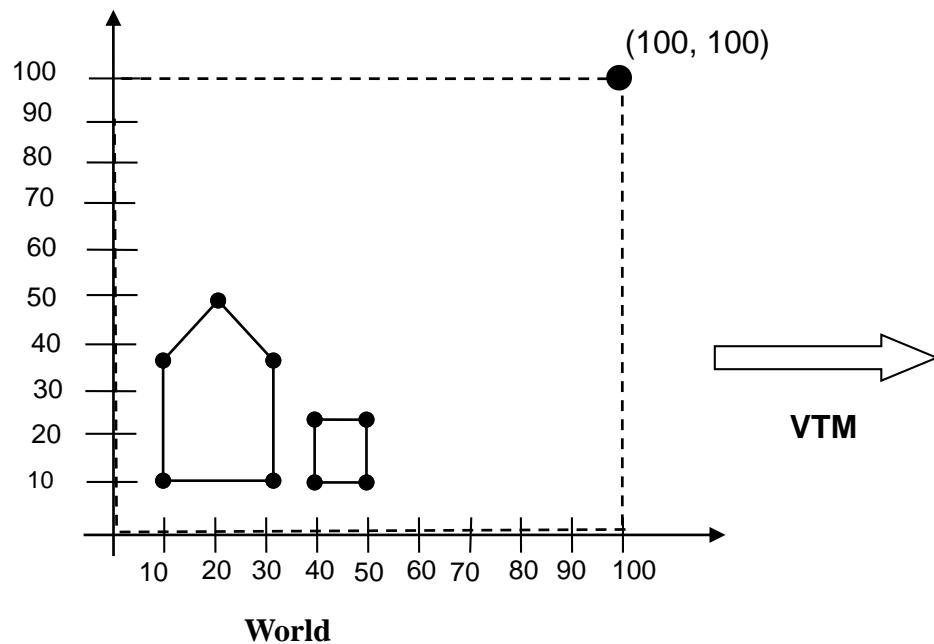
- Now we change window size *more*, recompute the VTM, and repaint again



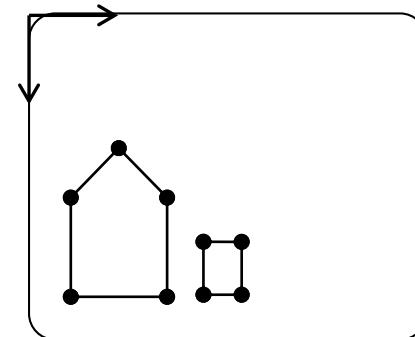
Display = ?

Changing Window *Location*

Suppose we start with this:



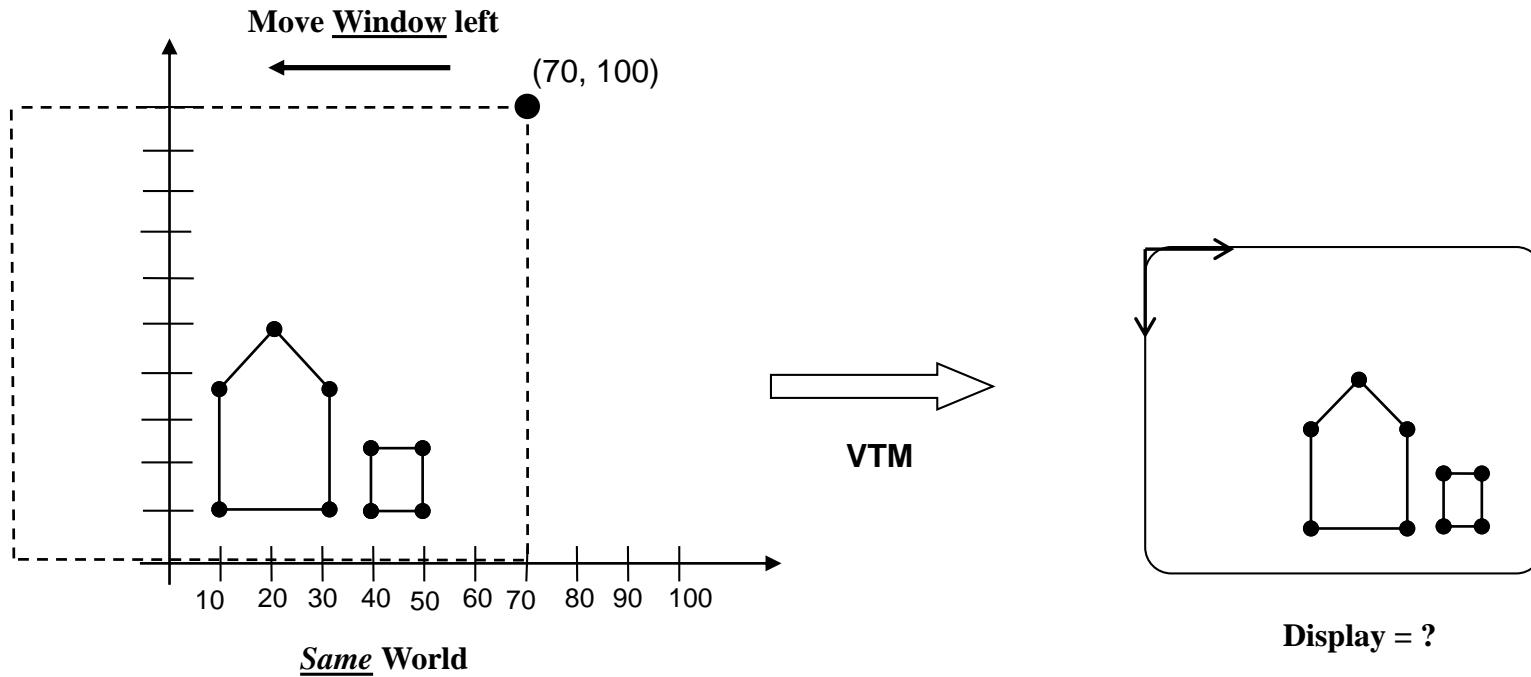
VTM



Display

Changing the Window Location (cont.)

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



$$X_{ND} = (X_w - W_L) * \frac{1}{(W_R - W_L)}$$

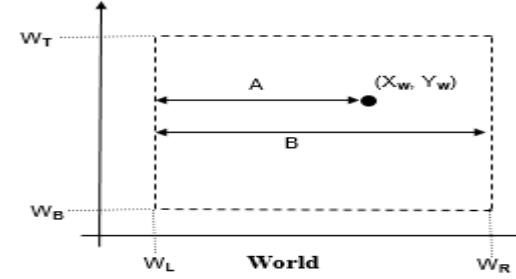
$$X_D = X_{ND} * \text{Scale}(DisplayWidth)$$

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
    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); // Make window larger
    }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); // Make window smaller
    }
    return true;
}
```

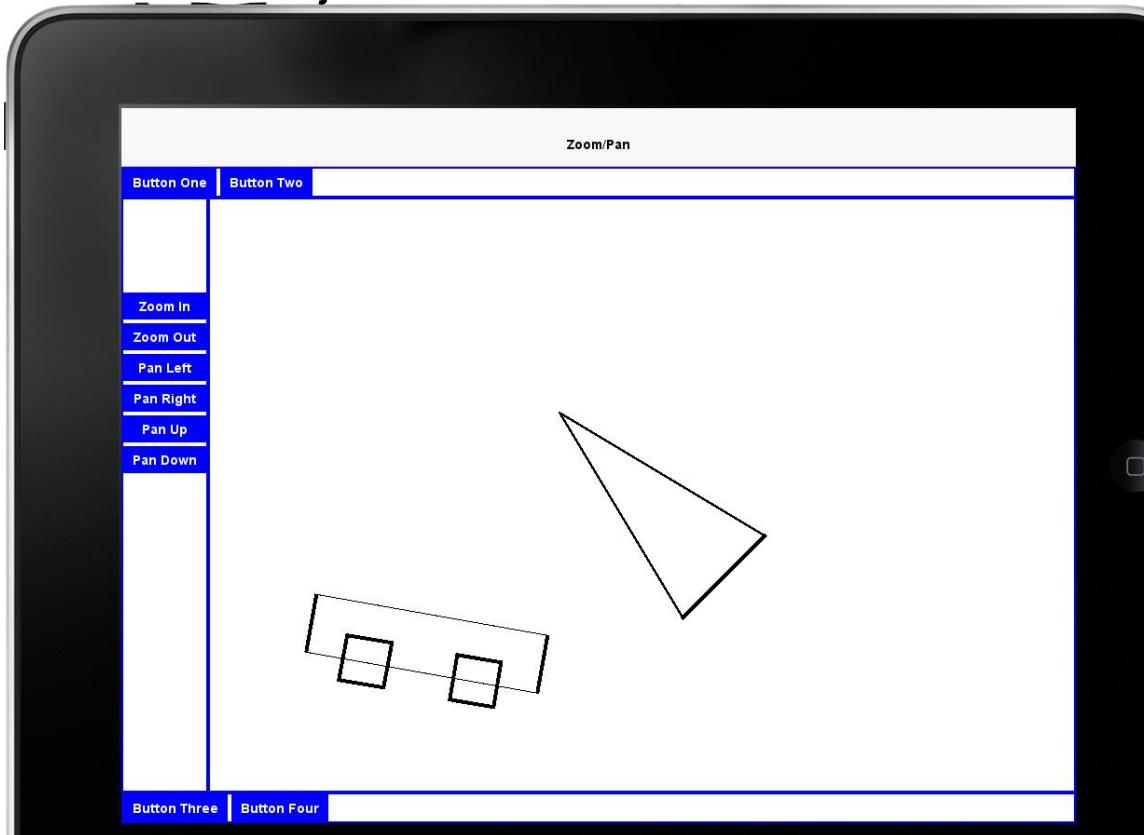
Pan with Pointer Dragging in CN1

```
/* 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); // i.e. positive delta would pan right (image would shift left)  
        else if (pPrevDragLoc.getX() > x)  
            panHorizontal(-5); // i.e. negative delta would pan left (image would shift right)  
        if (pPrevDragLoc.getY() < y)  
            panVertical(-5); // negative delta would pan down (image would shift up)  
        else if (pPrevDragLoc.getY() > y)  
            panVertical(5); // positive delta would pan up (image would shift down)  
  
    }  
  
    pPrevDragLoc.setX(x);  
    pPrevDragLoc.setY(y);  
}
```

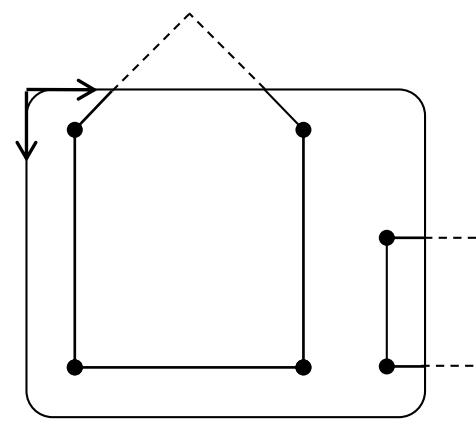
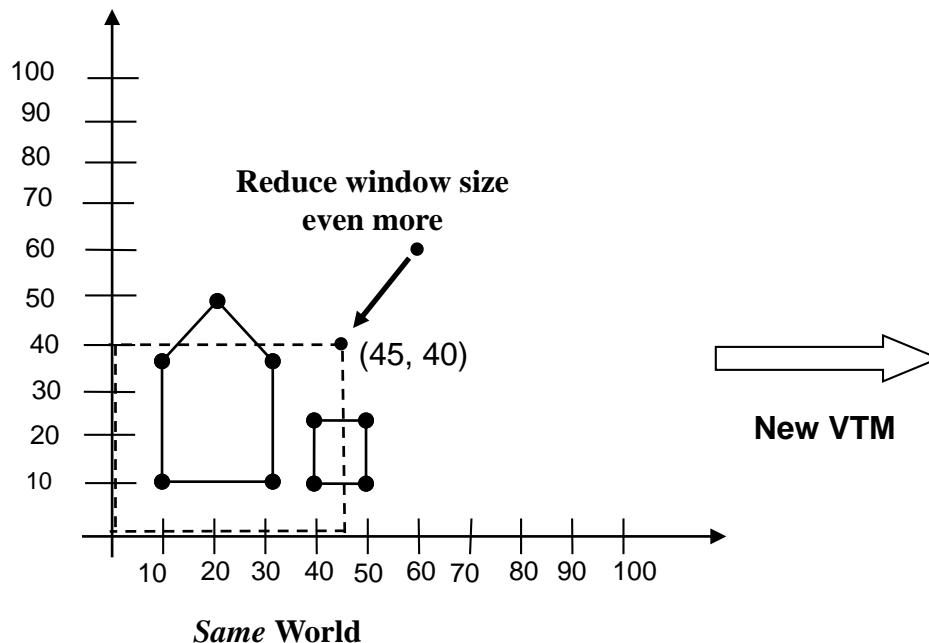
Zoom/Pan App ScreenShot

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**.



Recall: Changing the Window Size

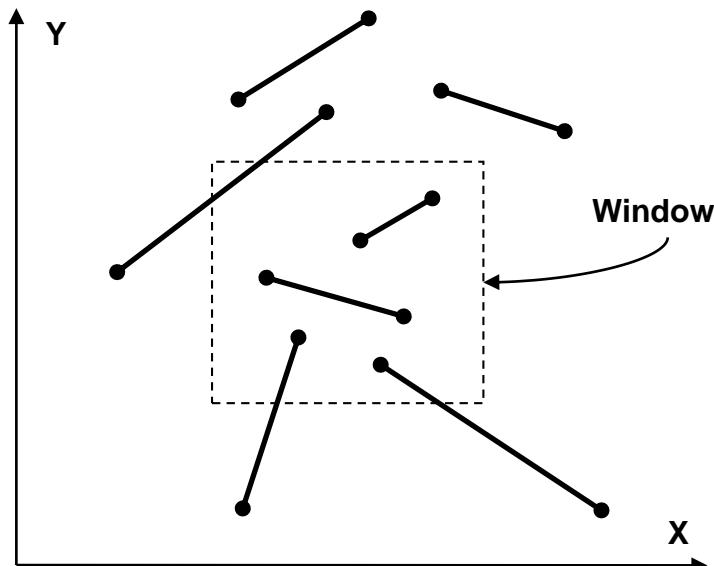
- Now we change window size *more*, recompute the VTM, and repaint again



Display = ?

Clipping

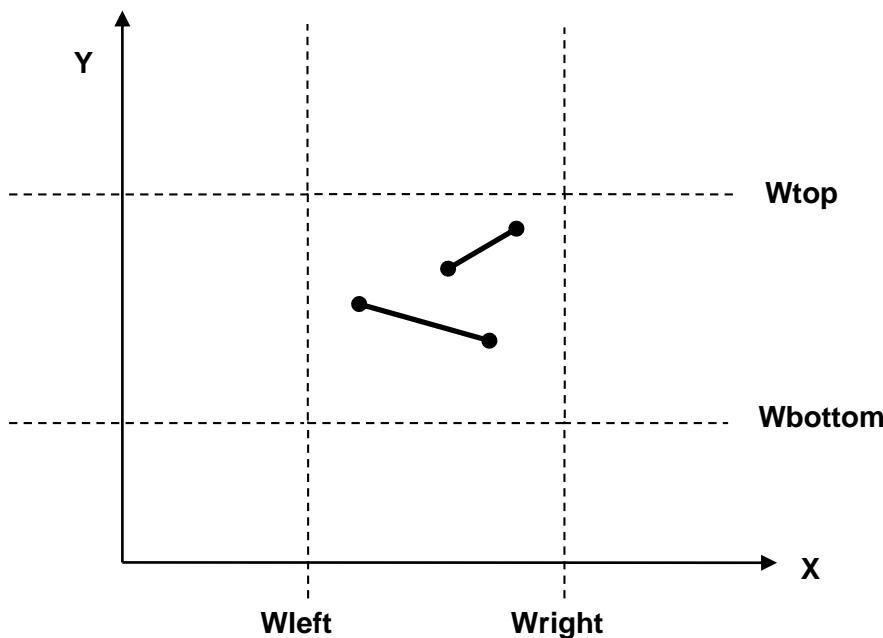
- Need to suppress output that lies outside the window
- For lines, various possibilities:
 - Both endpoints inside (totally visible)
 - One point inside, the other outside (partially visible)
 - Both endpoints outside (totally invisible ?)



Visibility Tests

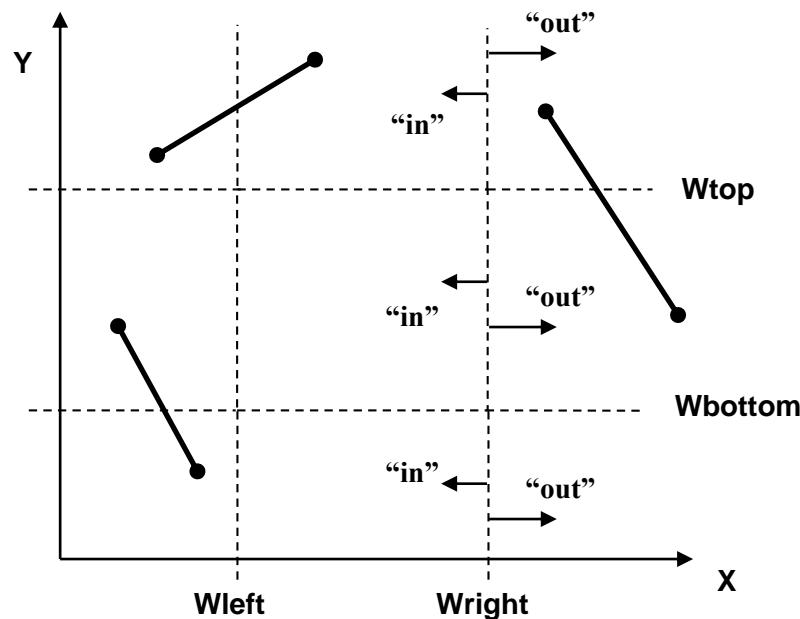
- “Trivial Acceptance”
 - Line is completely visible if both endpoints are:

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



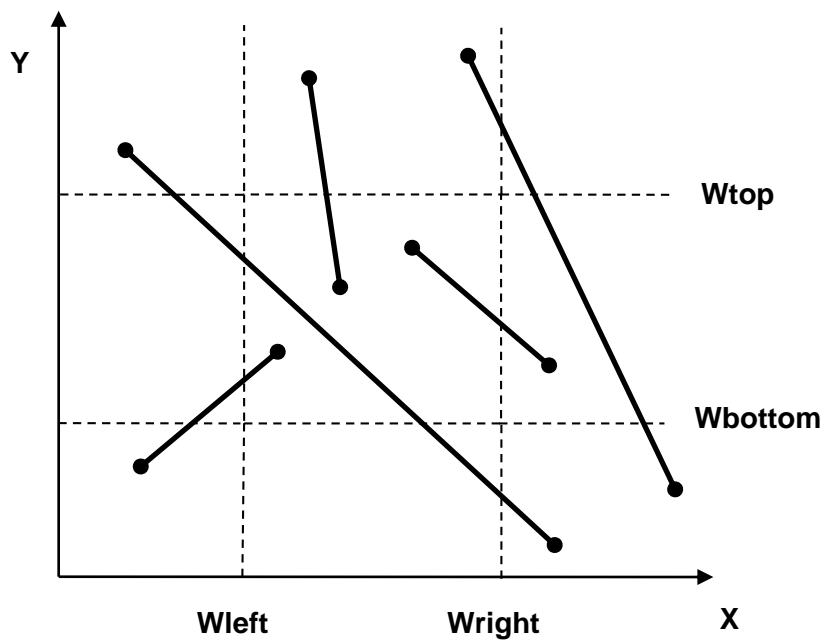
Visibility Tests (cont.)

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



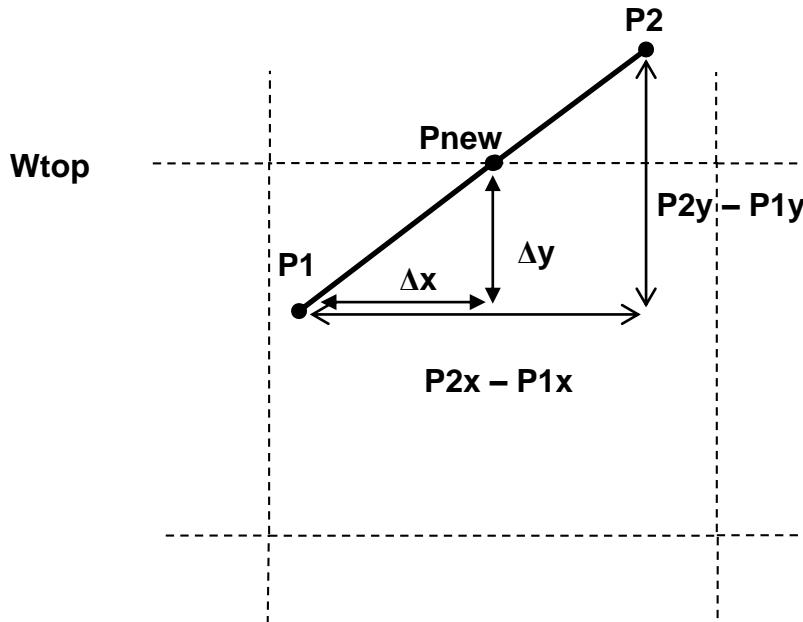
Visibility Tests (cont.)

- 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)



$$\text{Slope} = (P_2y - P_1y) / (P_2x - P_1x)$$

$$P_{newY} = W_{top}$$

$$\Delta y / \Delta x = \text{Slope}$$

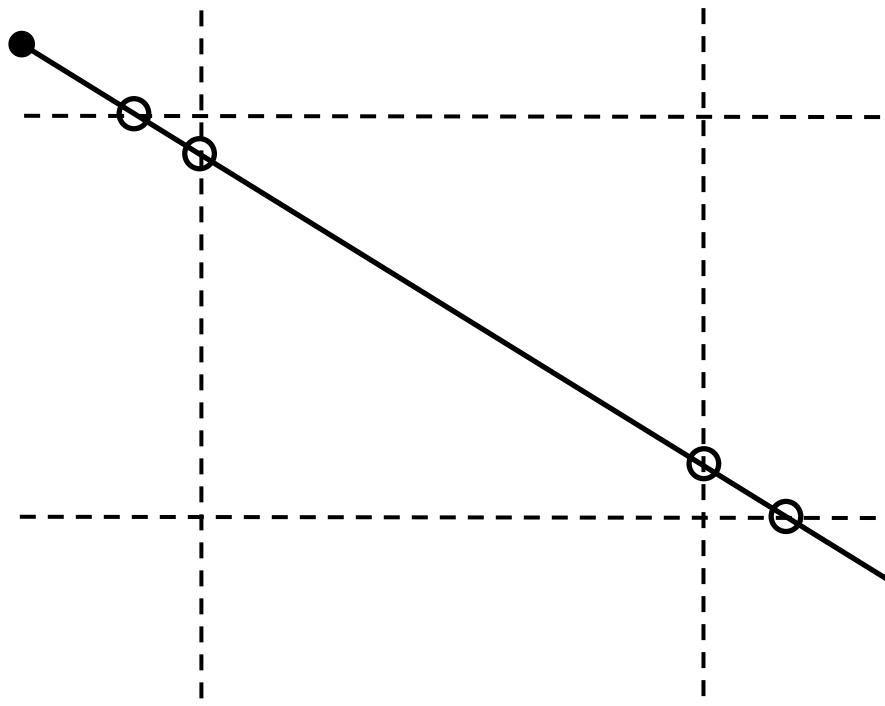
$$\Delta y = P_{newY} - P_1y$$

$$\Delta x = \Delta y / \text{Slope}$$

$$P_{newX} = P_1x + \Delta x$$

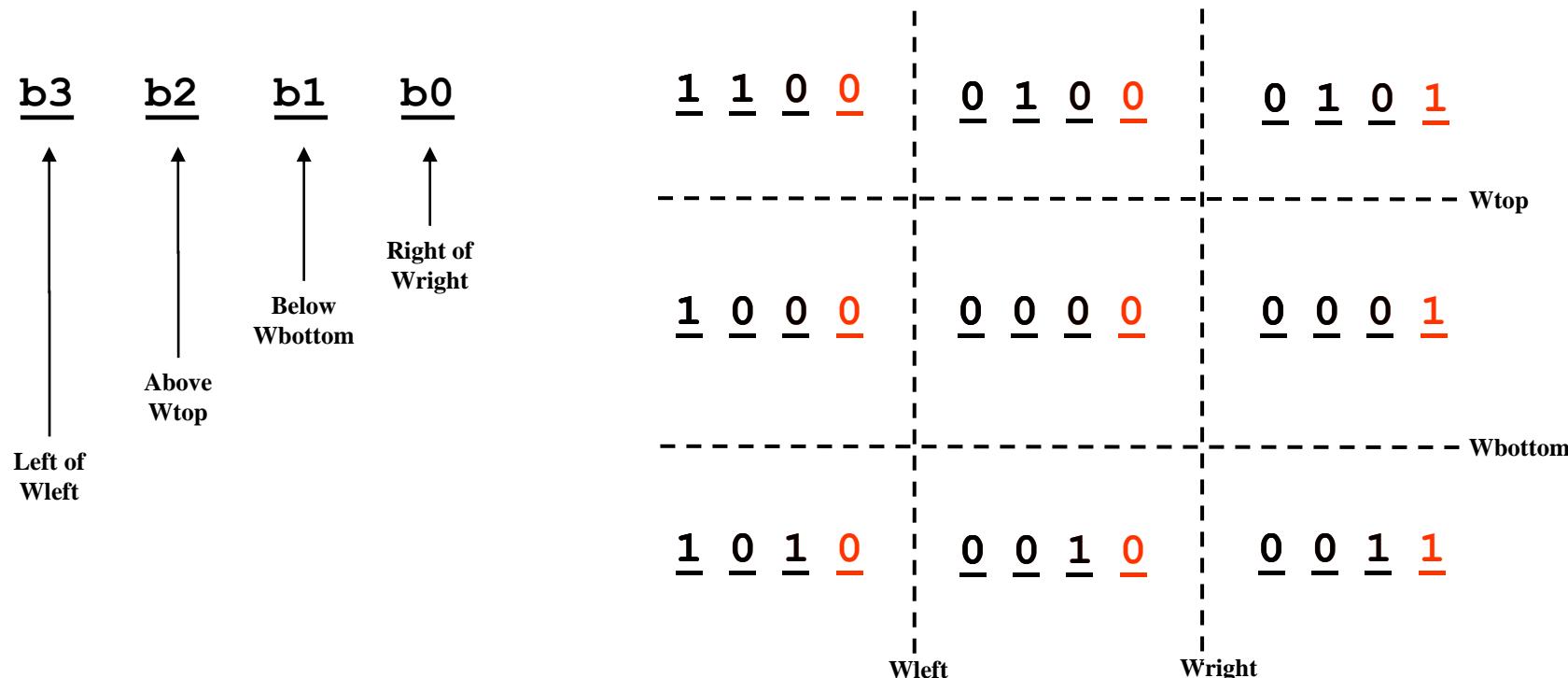
Clipping Non-Trivial Lines (cont.)

- 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) :	<u>b3</u>	<u>b2</u>	<u>b1</u>	<u>b0</u>
code(P2) :	<u>b3</u>	<u>b2</u>	<u>b1</u>	<u>b0</u>
<hr/>				
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
    }

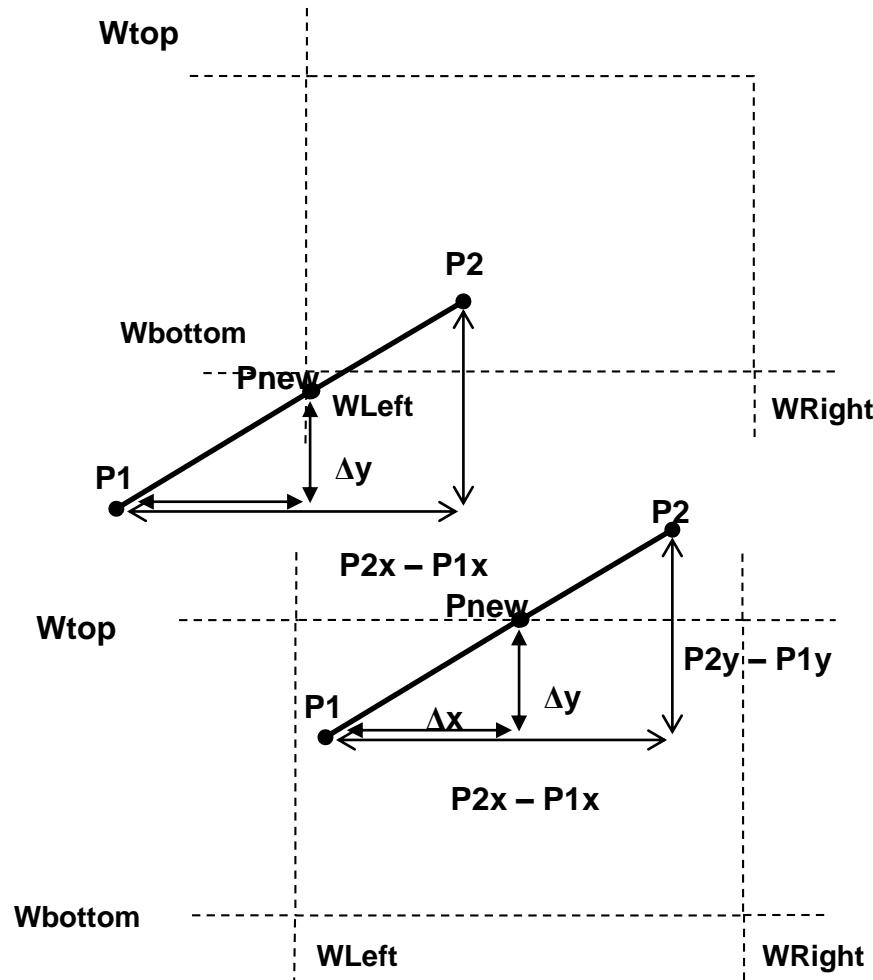
    return ( new Line(p1,p2) ); // the line is now completely inside the window
}
```

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

Rules for computing X/Y coordinates

- If line crosses Wleft or Wright then:
 - $x = Wleft$ or $Wright$
 - $y = y_1 + m(x - x_1)$
 - $m = \text{slope}$
- If line crosses Wbottom or WTop then:
 - $y = Wbottom$ or $Wtop$
 - $x = x_1 + (y - y_1) / m$
 - $m = \text{slope}$



Practice Problem

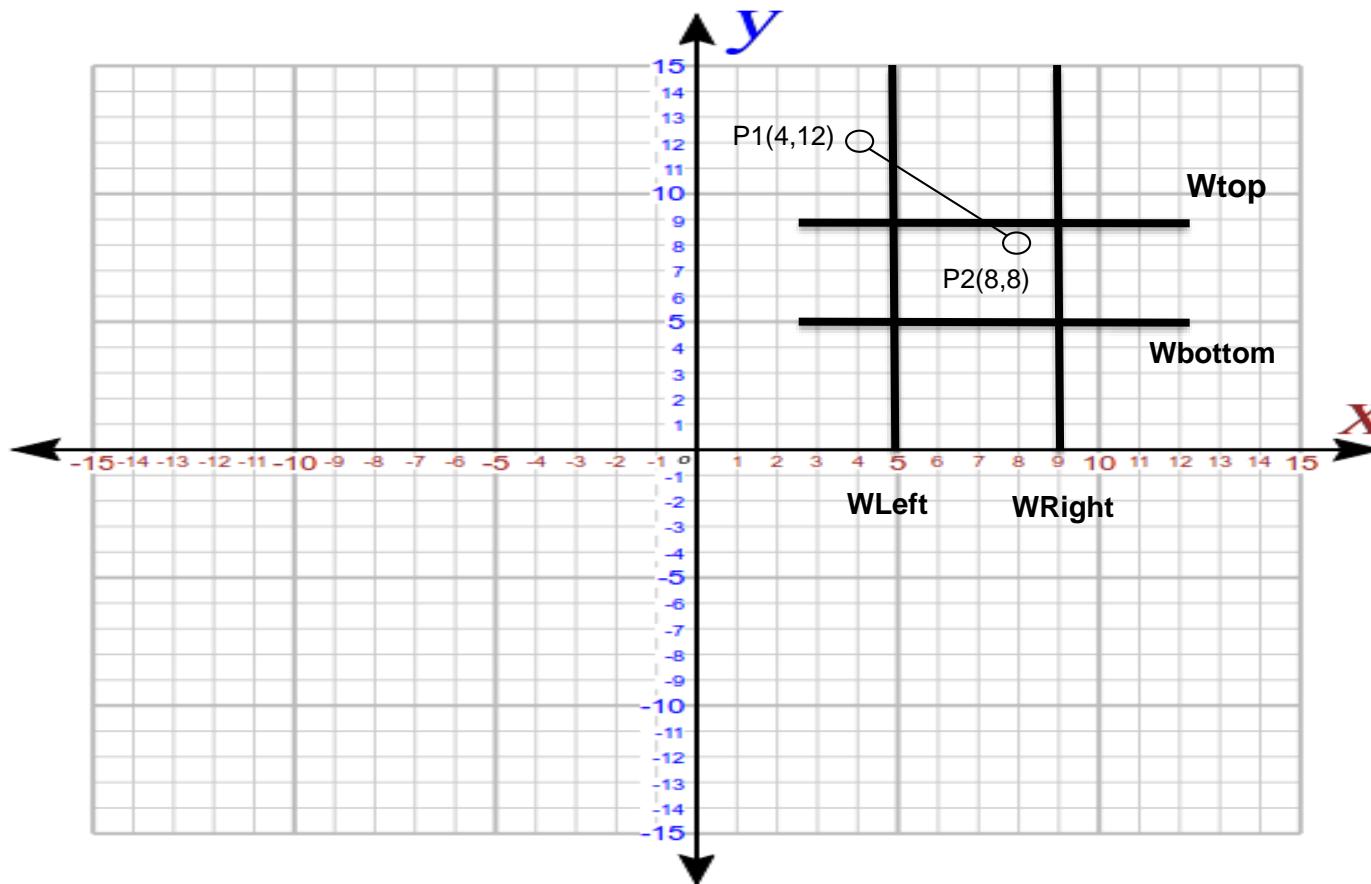
- A clipping window has the following property:
 - $\text{Window}(\text{left}, \text{right}, \text{bottom}, \text{top}) = (5, 9, 5, 9)$
 - A Line, L1, with the following end points is drawn in the word: P1 (4,12), P2 (8,8)
 - Show how the Cohen-Sutherland clipping algorithm will clip this line and what its final endpoints are?

Practice Problem (Cont)

- Draw window and points

Window(left,right,bottom,top) = (5,9,5,9)

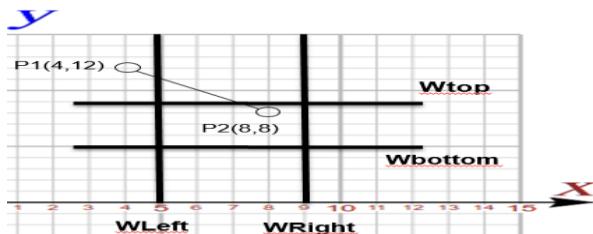
A Line, L1, with the following end points is drawn in the word: P1 (4,12), P2 (8,8)



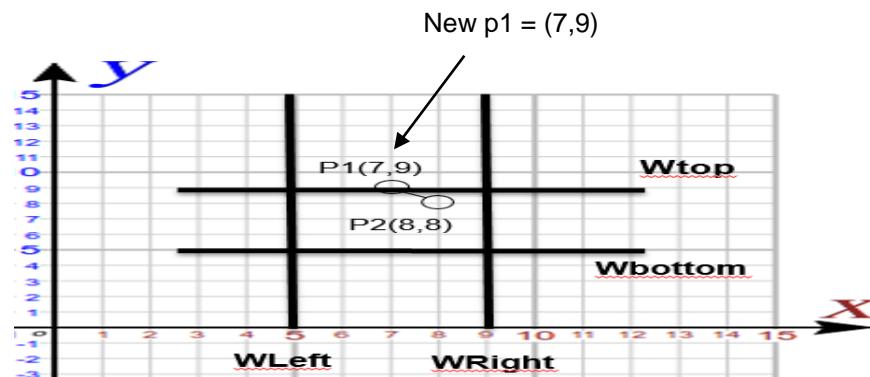
Practice Problem (Cont)

- Assign 4-bit CS codes for each input point: $c1=code(p1)=1100$, $c2=code(p2)= 0000$
- $(c1 \& c2) = 0000$ – Not Trivial rejection (Line is NOT completely invisible)
- $P1$ is not inside the window. Thus, no swapping.
- Compute new $p1$: $p1 = intersectWithWindow (p1, p2);$
 - line crosses $Wleft$ or $Wright$ then: $x = Wleft$ or $Wright$, $y = y1 + m(x-x1)$, $m=slope$
 - $x=5$, $y=12+1(5-4)=11$, slope=(12-8)/(4-8)=-1, $c1=code(p1) = 0100$
 - Line crosses $Wbottom$ or $Wtop$ then: $y = Wbottom$ or $Wtop$, $x = x2 + (y-y2)/m$, $m = slope$
 - $y=9$, $x=8+(9-8)/(-1)=7$, slope=-1 (same as above step)
 - New $p1 = (7,9) // \leftarrow$ New $P1$
 - $c1=code(p1) = 0000$

- $c1=code(\text{new } p1)=0000$

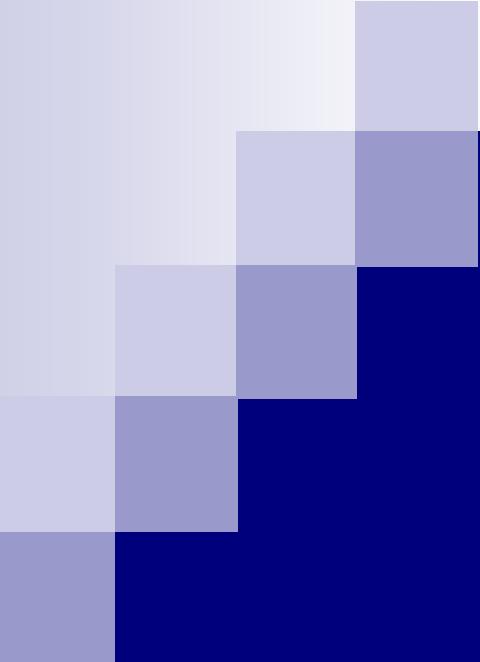


(Before Clipping)



(After Clipping)

- Now, both $c1$ and $c2$ are 0000, we return new line($p1, p2$)



17 – Publishing Codename One Application to Mobile Device

Computer Science Department
California State University, Sacramento

High level description

- Codename One allows Java developers to write native mobile apps for all devices.
- Once your CSC 133 - A3 assignment is complete, the most important thing is running the resulting app on your mobile devices.
- A build selected at the build server at codenameone.com and you choose your build entry.
- You can then either email the link to yourself using the dedicated button or just scan the QR code in the page. This will allow you to download and install the app to your device.
- Important: There is a charge for an application which exceeds certain size. Contact the Codename one for more information. There is no charge to test out the HelloWorld App.

Create Account with Codename One

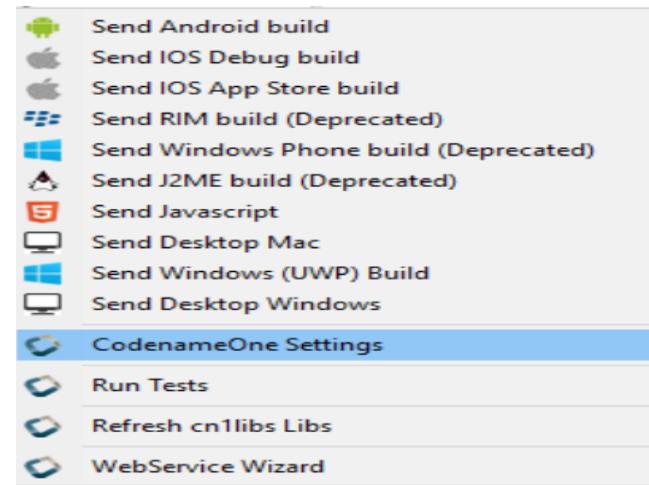
Step 1: Create a hello world project.

Step 2: Navigate

to www.codenameone.com/build-server.html and signup or login an account.

Generate Android Certification

Step 3: Right click the project
and select CodenameOne
Settings



Step 4: Click Android Certificate
Generator



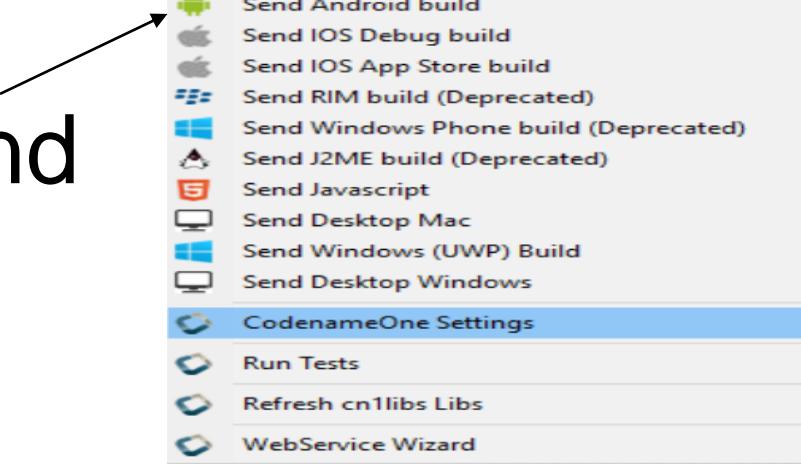
Android Certificate Generator

Simpler tool for generating Android certificates, unlike iOS certificates these are pretty easy to generate locally

Step 5: Fill out the Certification

Send Android Build

Step 6: Right click the project and select "Send Android build"



Download Android Build

Step 7: Navigate
to [www.codenameone.com/
build-server.html](http://www.codenameone.com/build-server.html)

Step 8: if the file is
successfully build, it will
look like the following.

Your Builds appear here. Notice that older builds are automatically deleted to preserve server space!

Android Successful build - 'HelloWorld' Took - 0:49 At - 15:55 Jan 17th 2019

Download Android Build (Cont)

Step 9: You can download it by scanning the QR code using your devices.

Your Builds appear here. Notice that older builds are automatically deleted to preserve server space!

Android Successful build - 'HelloWorld' Took - 0:49 At - 15:55 Jan 17th 2019

Downloadable Files	OTA Installation Files
 Starter-debug.apk	 Starter-debug.apk e-mail Link



Some Useful Resources

- <https://www.codenameone.com/how-do-i---create-a-basic-hello-world-application--send-it-to-my-device-using-eclipse.html>
- <https://www.codenameone.com/manual/index.html>
- <https://www.youtube.com/watch?v=pOLOoZFHxwU>

Study Guide — Final Exam

The Final Exam will be *closed book, closed notes, with NO Phone/Computer/Calculator allowed* except that you will be allowed to use a single ***hand-written*** sheet of 8.5x11" paper. Note that the sheet must be ***hand-written*** (by you); computer-printed or photocopied notes are not allowed. If you use a note sheet you will be required to put your name on it and to hand it in along with your exam.

To be well prepared for the Final Exam, you should have read all lecture notes, attended all lectures, completed all the reading and programming assignments due so far. In addition, you should be able to answer questions based on the topics listed below.

Object-Oriented Concepts

- (1) Understand the definitions of association, aggregation, composition and dependency, and give an example of each. Explain the similarities and differences between composition and aggregation.
- (2) Explain and use the elements of a UML class diagram, including general associations (using names, directions, and multiplicities), aggregation and composition, dependency, inheritance, and interfaces. Given a description of a program (set of classes), be able to draw the corresponding UML class diagram, and vice-versa.

Inheritance

- (3) Describe the purpose of inheritance (in the OO sense), and give an example. Include a description of the notions *superclass* and *subclass*. Tell how inheritance is expressed in Java/CN1 code, and explain the meaning and purpose of the keyword *super* in Java/CN1.
- (4) Explain what is meant by an *abstract method* in the general OO sense. Describe how abstract methods are implemented in Java/CN1, and what constraints exist on such methods, on the classes which contain them, and on other related classes.

Polymorphism and Interfaces

- (5) Explain the difference between the *apparent type* and *actual type* of an object, and be able to use and explain each of those notions correctly in Java/CN1 code.
- (6) Explain the difference between an abstract class and an interface in Java/CN1, and how Java/CN1 interfaces provide support for increased polymorphism in a program.

- (7) Know the usage of upcasting and downcasting.

Design Patterns

- (8) Explain the organization, purpose and context of each of the design patterns which have been discussed in class and in the assigned reading (e.g., **iterator**, **composite**, **singleton**, **observer**, **command**, **strategy**, **proxy**, **factory**, etc.). Give a specific example of using each design pattern. Give appropriate UML describing each design pattern and, for those design patterns which appear as part of the CN1 language definition, be able to explain the relevant CN1 interfaces and/or classes and how they are used.
- (9) Know how to implement each of the design patterns used in homework assignments, including assignments 1,2, and 3.
- (10) Explain what is meant by the “MVC architecture”. Include an explanation of the difference between an *architecture* and a *design pattern*.

Graphical User Interfaces and Event-Driven Programming

- (11) Be familiar with the event-driven operations of basic CN1 GUI components which have been discussed in class, including how events are generated by these components and how these events are handled by listeners that implement **ActionListener** interface. Describe two listener approaches: (i) a container which is an “event listener” for its own components, and (ii) separate listener object(s).

Interactive Techniques

- (12) Know the CN1 Graphic Class and its methods. Know how the component’s Repaint and Paint work. Drawing coordinates. Maintaining Graphic State. Implement Object selection. Review sample programs in Canvas.

Introduction to animation

- (13) Be able to explain “Frame based animation”? Utilize the CN1’s UITimer to schedule ticks to run drawing programs. “Self-animating” – Form consists of self-drawing objects. Compute animated coordination for a given current location. **Collision detection algorithms:** Detecting collisions (Bounding circle, Bounding rectangle), dealing with (responding to) collisions (Modify heading, Change appearance, etc). Know how to apply and use the ICollider, IDrawable, IMovable interfaces.

- (14) Be able to modify (extend) existing collision detection algorithms to address a very simple a collision avoidance problem.

Introduction to Sound

- (15) Know the popular sound APIs, Playing sounds in CN1. Create background sound that loops.

Transformation

(16) Explain what transformation is. Affine Transformations: Translation, Rotation, Scaling, Reflection. Matrix Representation of Transforms. Translate, rotate scale and reflect objects using matrices.

(17) Applications of Affine Transforms: CN1 Coordinate Systems and Transforms, Local Coordinate Systems, Display-mapping Transforms, Transformable Objects, Composite Transforms, Hierarchical Object Transforms, Dynamic Transforms.

Apply these techniques to build a new Hierarchical object such as a Windmill (i.e. with multiples instances in various sizes). Please refer to a demonstration in lecture.

(18) Viewing Transformation: World & Display Coordinate Systems; World-to-Display Mapping, World Window, and the Viewing Transformation (VTM); Zoom and Pan; Display-to-World Mapping; Clipping (Cohen-Sutherland algorithm).

Practice the clipping exercise per lecture material.

Study Guide — Midterm Exam Tentative List (3/01/19)

The Midterm Exam will be *closed book, closed notes*, except that you will be allowed to use a single ***hand-written*** sheet of 8.5x11" paper. Note that the sheet must be ***hand-written*** (by you); computer-printed or photocopied notes are not recommended. If you use a note sheet you will be required to put your name on it and to hand it in along with your exam.

To be well prepared for the Midterm Exam, you should have read all lecture notes, attended all lectures, completed all the reading, attendance quizzes, and programming assignments due so far. In addition, you should be able to answer questions based on the topics listed below.

Object-Oriented Concepts

- (1) Understand and correctly use the basic Java/Codename One (CN1) elements that we have utilized in class such as objects, access modifiers, constructors, references, arrays, vectors, parameter passing, etc. Understand relevant aspects of the compile-execute environment such as built-in frameworks (e.g. CN1's UI package).
- (2) Explain the difference between a class and an object (note that this is not Java/CN1-specific).
- (3) List the four components generally recognized as comprising the “Object-Oriented” (OO) programming paradigm. Give a general description of their meaning and be able to give examples of their application.
- (4) Explain what is meant by “information hiding” and “bundling”, and give an example.
- (5) Understand the definitions of association, aggregation, composition and dependency, and give an example of each. Explain the similarities and differences between composition and aggregation.
- (6) Explain and use the elements of a UML class diagram, including general associations (using names, directions, and multiplicities), aggregation and composition, dependency, inheritance, and interfaces. Given a description of a program (set of classes), be able to draw the corresponding UML class diagram, and vice-versa.
- (7) Explain the notion of “accessor” (including “mutators”), and how they relate to the OO notions of abstraction and encapsulation. Be able to explain why providing public accessor methods is not equivalent to making fields public.
- (8) Show how to construct a set of classes which have associations between them so that fields are private and yet the classes can reference each other and use accessors appropriately.
- (9) Explain what is meant by *overloading* a method, and give an example. Explain why constructors are commonly overloaded.

Inheritance

- (10) Describe the purpose of inheritance (in the OO sense), and give an example. Include a description of the notions *superclass* and *subclass*. Tell how inheritance is expressed in Java/CN1 code, and explain the meaning and purpose of the keyword *super* in Java/CN1.
- (11) Give examples of the “is-a” and “has-a” relationships, and explain their difference.
- (12) List three major uses (purposes) of inheritance and give an example of each.
- (13) Explain what is meant by *overriding* a method, including how it differs from *overloading*.
- (14) Explain what is meant by *multiple inheritance*, including its advantages and disadvantages.
- (15) Explain the OO notion of separation of interface from implementation. Describe how this is carried out in Java/CN1, and its relationship to multiple inheritance.
- (16) Explain what is meant by an *abstract class* in the general OO sense. Describe how abstract classes are implemented in Java/CN1 and what constraints exist on such classes and on other related classes.
- (17) Explain what is meant by an *abstract method* in the general OO sense. Describe how abstract methods are implemented in Java/CN1, and what constraints exist on such methods, on the classes which contain them, and on other related classes.

Polymorphism and Interfaces

- (18) Give examples of both predefined CN1 interfaces and user-specified interfaces.
- (19) Explain what is meant by polymorphism in a programming language. Give examples of the application of polymorphism in a program.
- (20) Explain the meaning of the term “*dynamic (late) binding*”, and how it applies in Java/CN1.
- (21) Explain the difference between the *apparent type* and *actual type* of an object, and be able to use and explain each of those notions correctly in Java/CN1 code.
- (22) Explain the difference between an abstract class and an interface in Java/CN1, and how Java/CN1 interfaces provide support for increased polymorphism in a program.

Displays and Color

- (23) Describe how the abstraction “color” is implemented in CN1. Include a description of the notion of the “RGB color cube”.

Design Patterns

- (24) Explain what is meant by the term “design pattern”. List the three major categories of design patterns, and give an example of each.
- (25) Explain the organization, purpose and context of each of the design patterns which have been discussed in class and in the assigned reading (e.g., *iterator*, *composite*, *singleton*, *observer*, *command*, *strategy*, *proxy*, *factory*, etc.). Give a specific example of using each design pattern. Give appropriate UML describing each design pattern and, for those design patterns which appear as part of the CN1 language definition, be able to explain the relevant CN1 interfaces and/or classes and how they are used.
- (26) Know how to implement each of the design patterns used in assignments.
- (27) Explain what is meant by the “MVC architecture”. Include an explanation of the difference between an *architecture* and a *design pattern*.

Graphical User Interfaces and Event-Driven Programming

- (28) Be able to write CN1 code showing the basic steps involved in building a Graphical User Interface consisting of components such as labels, buttons, checkbox, text field, side/overflow menu items, title bar menu item, and containers.
- (29) Explain the purpose of a “Layout Manager”, and be able to describe the general operation of at least two CN1 layout managers.
- (30) Be familiar with the event-driven operations of basic CN1 GUI components which have been discussed in class, including how events are generated by these components and how these events are handled by listeners that implement **ActionListener** interface. Describe two listener approaches: (i) a container which is an “event listener” for its own components, and (ii) separate listener object(s).
- (31) Explain the relationship between the CN1 **Command** and **Button** classes and the elements of the Command design pattern.
- (32) Explain what key bindings are and how they are implemented in CN1.

CSC 133: Object-Oriented Computer Graphics Programming – Spring 2019

TENTATIVE COURSE OUTLINE

Week	Topics	Readings and Assignments
1	Course Introduction and Overview Introduction to Mobile App Development and CN1 OOP Concepts: Abstraction; Encapsulation (Classes & Accessors); UML Class Diagrams, Class Associations (Aggregation and Composition)	CN1: Ch 1 Horstmann: Ch 1 (all); Ch 2 Sec 2.1–2.8; Ch 3 Sec 3.1–3.4; Ch 7 Sec 7.1 – 7.4
2	Inheritance: Inheritance Hierarchies; Overriding and Overloading; Extension vs. Specialization vs. Specification; Abstract Classes and Methods; Single vs Multiple Inheritance	Horstmann: Ch 6 Sec 6.1, 6.3, 6.8, 6.9 A#1: Class Associations
3	Polymorphism: Types of Polymorphism (Static vs Dynamic); Upcasting and Downcasting Interfaces: Interface Hierarchies; Interface Subtypes, Interfaces and Polymorphism; Abstract Classes vs Interfaces; Multiple Inheritance via Interfaces	Horstmann: Ch 3 Sec 3.5; Ch 4 Sec 4.1–4.5
4	Design Patterns: Creational vs. Structural vs. Behavioral; Singleton, Composite, & Iterator Patterns; Observer Pattern; MVC Architecture	Horstmann: Ch 5 Sec 5.1-5.3, 5.5; Ch 10 Sec 10.5
5	GUI Basics: Display Devices & Color; Components in CN1 User Interface (UI) package (Component, Container, Form, Dialog, Label, Button, Checkbox, TextField, SideMenu); Layout Managers and Strategy Pattern	Schaum: Ch 1; Ch 2 Sec 2.1-2.4; CN1: Ch 2; Ch 5 Horstmann: Ch 5 Sec 5.4.3, 5.7, 5.8 A#2: Design Pats. & GUIs
6	Event-driven Programming: CN1Action Handling (ActionEvent, ActionListener, Command); CN1 Key Handling; Command, Factory and Proxy Patterns	CN1: Ch 6 Horstmann: Ch 5 Sec 5.3 Ch 10 Sec 10.2, 10.3, 10.4
7	Interactive Techniques: CN1 Graphics and Component Repainting, Pointer Handling; Object Selection	CN1: Ch 6
8	Introduction to Animation: Frame-based Animation; CN1 UITimer, Animation via Image Movement; Collision Handling <u>MIDTERM EXAM</u>	A#3: Interactive Graphics & Animation
9	Introduction to Sound: Sound Files, Sound APIs, CN1 MediaManager and Media Transformations: Vectors and Matrices, Affine Transformations, Matrix Representation, Homogeneous Coordinates, Concatenation of Transforms	Schaum: Appendix 1; Ch 4 Secs 4.1-4.3

10	Applications of Affine Transforms: CN1 Coordinate Systems and Transforms, Local Coordinate Systems, Display-mapping Transforms, Transformable Objects, Composite Transforms, Hierarchical Object Transforms, Dynamic Transforms	CN1: Ch 6 Schaum: Ch 4 Sec. 4.4
11	Viewing Transforms: World & Display Coordinate Systems; World-to-Display Mapping, World Window, and the Viewing Transformation (VTM); Zoom and Pan; Display-to-World Mapping; Clipping	Schaum: Ch 5 Sec 5.1–5.3, 5.5 A#4: 2D Transforms
12	Lines and Curves: Rasterization, DDA & Bresenham Algorithms; Parametric Line and Curve Representation; Bezier Curves	Schaum: Ch 3 Sec 3.1, 3.2, 3.8
13	Threads: Java/CN1 Thread and Runnable; Synchronization	Horstmann: Ch 9 (all)
14	Device Installation and Code Signing: Installation of CN1 Apps to Physical Mobile Devices and Code Signing for iOS and Android Devices	CN1: Ch 14
15	Additional Topics As Time Permits: Adapter, Decorator, Memento; Review for Final; Introduction to Related CSC Courses	Horstmann: Ch 5 Sec 5.6; Ch 10 Sec 10.1, 10.7
16	Final exam	

Text for the Readings:

- **Horstmann:** Object-Oriented Design & Patterns, 2nd Ed., by Cay Horstmann; Wiley; ISBN 0-471-74487-5
- **Schaum:** Schaum's Outlines Computer Graphics, 2nd Ed., by Xiang & Plastock; McGraw-Hill; ISBN 0-07-135781-5
- **CN1:** Codename One Developer Guide
(<https://www.codenameone.com/files/developer-guide.pdf>)

CSC 133: Object-Oriented Computer Graphics Programming – Spring 2019

COURSE SYLLABUS

Instructor: Dr. Doan Nguyen

Office Hours: M: 9:00AM-12:00 PM and by appointment

Office: Riverside Hall 5009

Telephone: (916) 278-6834

E-Mail: doan.nguyen@csus.edu

Course Materials: Syllabus, outline, notes, and other materials are available at the "Files" section of Canvas

Important reminder:

During class time, please keep your cell phones turned off and please refrain from browsing, social networking, gaming, messaging etc.

Catalog Description:

An introduction to computer graphics and to advanced topics in object-oriented (OO) programming. The OO paradigm is used throughout, utilizing computer graphics as the vehicle for solidifying basic OO concepts, studying the implementation of event-driven systems, and for developing a thorough understanding of advanced OO concepts such as inheritance and polymorphism. Topics include fundamental concepts of object-oriented programming, software design patterns, graphic devices, line and surface drawing, simple 2D and 3D representation, and use of User Interface components. Prerequisites: CSC 130 and CSC 131. Units: 3

Prerequisites:

CSC 130 – Algorithms and Data Structures & CSC 131 – Introduction to Software Engineering.

You must have already **completed** both CSC 130 and CSC 131 (or their equivalents at another school), each with a grade of C- or better, in order to take CSC 133. In addition, you must have also completed CSC 15, CSC 20, CSC 28, and Math 29 (or their respective equivalents at another school), each with a grade of C- or better, since those courses are prerequisites for CSC 130 and/or CSC 131. Student records will be reviewed to determine whether a student has taken the required prerequisites and if not (s)he will be dropped from the class.

Completion of Math 30, Math 100, and Physics 11A will be very helpful, although not required.

Prerequisites by Topic:

- Three semesters (minimum) experience programming in a high-level language (C, C++, Java, Python, etc.);

- Experience with “Object-based” programming – class definitions, object instantiation, method invocation, public vs. private fields, etc.;
- Implementation of linear lists, including stacks & queues, and of binary trees; use of recursion in a program;
- Familiarity with UML class diagrams and their use in specifying relationships including aggregation, composition, and inheritance;
- Pre-calculus math including trigonometric functions, Cartesian coordinates, points, lines, and planes in space, coordinate transformations, conics, algebraic relations and functions, polynomial equations, inequalities, and matrix operations.

Repeat Policy:

Department policy specifies that students may not repeat a Computer Science course **more than three times** (that is, take a course for **a fourth – or subsequent – time**). Any student who wishes to repeat a course more than three times must submit a petition requesting permission to do so. Student records will be reviewed to determine whether a student is taking this course for a forth (or subsequent) time. Any such student must return an **approved** petition to the instructor **within the first two weeks of class**. Any student who does not submit an approved petition will be dropped from the class. Petitions are available in the Department office (Riverside Hall 3018) and require the signature of both the Instructor and the Department Chair.

Course Structure:

This course has two separate but equally important two main goals: an understanding of the advanced features of the so-called “object-oriented (OO) programming paradigm”, and an understanding of the fundamentals of computer graphics (CG) programming. This course also allows students to gain experience in mobile application development by applying the introduced OO and CG concepts to this development environment.

We will cover the OO concepts of abstraction, encapsulation, inheritance, and polymorphism, including discussion of their variations both conceptually and in terms of language-specific implementations. We will also look at formalisms for specification of OO systems (specifically, Unified Modeling Language or UML). In addition, we will cover various design patterns for OO systems which will be emphasized as an underlying theme throughout the course.

In the CG area we will cover hardware characteristics of graphics systems, and go through basic CG operations such as line and polygon drawing, 2D object modeling, geometric transformations, and transformations for mapping between “world” and “screen” space. We will also cover CG-based user interfaces (GUIs), including how event-driven components for GUI implementation work.

The OO and CG topics in the course will not be covered separately, but rather will be closely integrated and frequently co-mingled. It will be quite common to spend one class period (or even just part of one period) talking about some OO-related topic, and then switch to a discussion of a CG-related application of that OO topic. One reason for this is to insure equal

emphasis on both of the course goals; another is simply that many of the OO topics have a strong relationship to various CG topics. Covering the topics in parallel thus has the extra benefit of reinforcement.

There is a potential drawback to organizing a course as just described: students who regularly **miss class**, or who habitually **come in late**, will likely find that the penalty for doing so is significantly greater than it might be in some other courses. You might find, for example, that a graphics concept is being explained by utilizing an OO concept previously explained (but which was missed due to lateness or non-attendance). This makes it considerably more difficult to understand the new topic, which in turn tends to have negative impact on both the amount of time it takes to do assignments and on your ability to do well on exams. The course organization described above should make it easy for students who attend class regularly and on time to keep up and do well.

This course utilizes mobile technology as a tool for teaching course topics and requires students to solve their assignments using this emerging technology. These allow students to relate course topics (e.g. OO programming, design patterns, graphical user interface design, event handling, animation, and computer graphics techniques) to mobile application development process, a skill that is increasingly demanded by the job market. Since the prerequisite course sequence at CSUS uses the Java programming language, as a mobile application development environment we will be using a Java-based framework called Codename One (CN1). Other advantages of CN1 includes: it is a cross-platform framework (i.e. the applications developed in this framework run on various mobile platforms including Android, iOS, and Windows); it is free for academic use and it is open-source; and it comes with a simulator environment (i.e. to develop and run the application you do not need to have a physical mobile device).

Texts and References:

The following text materials are **required**:

CSC 133 Lecture Note Slides, Spring 19: available at the “Files” section of Canvas

Codename One Developer Guide - Revision 5.0: available at:

<https://www.codenameone.com/files/developer-guide.pdf>

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

The following text materials are **recommended**:

Object-Oriented Design & Patterns, 2nd Ed., by Cay Horstmann; Wiley; ISBN 0-471-74487-5

Schaum's Outlines Computer Graphics, 2nd Ed., by Xiang & Plastock; McGraw-Hill; ISBN 0-07-135781-5

They are available in the Hornet Bookstore as well as numerous on-line booksellers.

Supplemental (online) materials:

Basic Debugging With Eclipse: <https://www.youtube.com/watch?v=PJWtO5wrptg>

Basic functions for using the Eclipse debugger with Java, such as breakpoint, step in/over functions, changing variable values, etc.

Since the mobile application development environment we will be using in this course (i.e. Codename One) is a Java-based framework, students not already familiar with Java may want to acquire a book on Java. A separate bibliography of popular Java texts, as well as a bibliography on “Java for C++ Programmers” topics, is available from the instructor. Some time will be spent during the semester focusing on important differences between Java and C++; however, these discussions will not be sufficient to comprise a tutorial on Java; it will be the responsibility of each student to be proficient in using the basic constructs of Java (this should not be a problem for students with solid background in object-based programming in C++).

Assignments:

Throughout the semester, programming assignments will be given which are not necessarily weighted equally. They are required to be solved with Java-based mobile application development environment called Codename One. Assignments will be cumulative; thus students should not skip an assignment.

Late assignments will be accepted (the instructor would rather have you do the work late than not at all) up until **ONE WEEK** past the original due date, but **a penalty of 5% per day will be applied to all late work.** Hence, assignments submitted one week after the deadline (and beyond) will not be accepted and the maximum late penalty will be 50%. School holidays and weekends will be counted as regular days in computing lateness of assignments, so for example if an assignment was due on a Friday and was submitted on a Monday, it would be counted as three days late.

Assignment submissions will be done using Canvas; submission time as recorded in Canvas is considered the time at which assignments are submitted. Assignment submissions can be replaced (updated) prior to the due date, but **assignments which have been submitted cannot be replaced once the due date has passed.** Technically, Canvas allows you to submit new versions indefinitely. However, the abovementioned assignment submission policy dictates that 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). Hence, if you are planning to do a late submission, you should not submit a version of your assignment before the due date (because if you do a submission before due date, versions submitted after due date will be ignored). Also, you should submit your late submission when it is ready (because after you submit the first late version, the submissions you make later will be ignored).

Exams:

There will be a midterm exam and a final exam. Study guides will be provided before the exams. However, you are responsible for knowing all the content of CSC 133 Lecture Note Slides. Makeup exams are not given except under the most extreme (and documented) circumstances, in which case every effort should be made to contact the instructor in advance.

Final exam schedule:

Spring 2019 Day Classes

Class Day(s)	Class Start Time From	Class Start Time To	Spring Exam Day	Exam Time
TR	9:00am	10:29am	Tues., May 14	10:15am - 12:15pm
TR	1:30pm	2:59pm	Thurs., May 16	12:45pm - 2:45pm

Grading:

The following scale is used in determining the grades:

Range	Letter Grade
94-100	A
90-93	A-
87-89	B+
84-86	B
80-83	B-
77-79	C+
74-76	C
70-73	C-
67-69	D+
64-66	D
60-63	D-
59 or Less	F

Overall course grades will be computed using the following policy:

Assignments (including attendance)	40%
Midterm Exam	20%
Final Exam	25%
Quizzes	10%
Roll Attendance	5%

In addition to grades being computed as described above, one further constraint applies: **in order to achieve a passing grade of at least C- for the course, it is a requirement that you achieve at least passing completion (that is, D- or better) of (1) the average of Programming Assignments; (2) the average of the exams (midterm and final exams); and (3) Attendance & Quizzes.**

At the end of the semester, a final score of each student will be calculated according to the above policy. These scores **MIGHT** be curved based on final scores of all students in the class to determine the final grades. A positive curve will be used. That is, the grading scale listed above indicates the minimum grade that a student will receive based on his/her un-curved final score.

Roll will be taken without announcing, and the attendance grade will be proportional to the number of classes that the student is present. The student will be counted as absent for the class that (s)he does not attend unless the instructor is notified (in advance) that there is an unusual (and documented) circumstance.

Students are required to keep backup (soft) copies of all submitted work until after final grades are posted.

Computers & Communication:

There are several computer options available for doing class assignments. Students may work on any school machine onto which Eclipse IDE for Java Developers and Codename One (CN1) have been installed or on their personal machine provided that they install the abovementioned software.

Eclipse IDE for Java Developers (Neon version) and CN1 are installed to 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], and ECS Windows Terminal Server called Hydra. Instructions for reaching Hydra terminal can be found at https://www.ecs.csus.edu/computing/help_docs/Connecting_to_Hydra-Windows_and_Unix.pdf. For better performance please use lab machines or your own machine instead of remotely logging into Hydra.

CN1 installation instructions can be found at <http://www.codenameone.com/download.html>. CN1 is installed as a plugin to one of the following Integrated Development Environments (IDEs) which run on various operating systems: Eclipse, NetBeans, or IntelliJ IDEA (all of which are free). However, for solving the CSC 133 assignments, the students are **required** to use CN1 plugin which is installed to “**Eclipse IDE for Java Developers**” (available at <http://www.eclipse.org/downloads/>). In addition, the instructor **recommends** running **Windows** as the operating system. Please note that prior to installing the Eclipse and CN1, the students must install latest version of Java SE Development Kit (JDK) - version 8 - which is freely available at <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>.

If you choose to work on the assignments on your personal machine, prior to submitting them, you are **strongly encouraged** to build and test them on one of the lab machines and then submit the files tested on the lab machine. Since the assignments will be graded on a lab machine, this step will ensure the validity of your submission (e.g., if your personal machine is running Linux, MacOS or a different version of Windows other than Windows 7, this step will make sure that your assignments also work as expected on a lab machine which runs Windows 7). To build and test your assignment in the lab, you can copy your assignment directory located in the Eclipse workspace directory of your machine to the Eclipse workspace directory of the lab machine and use “File -> Import -> General -> Existing Project into Workspace” option of Eclipse to import your project to the lab workspace.

Regardless of which computer option you choose for doing class assignments, you are expected to be familiar with and regularly read “Discussions” section of Canvas. It is assumed that notices sent to the “CSC 133 – Communication Forum” located under “Discussions” are received and read by you (Tip: You can use “Subscribe” functionality of the forum to get a

notification e-mail sent to your SacLink account whenever a new notice is posted to the forum.). You need to have SacLink account to login to Canvas. If you do not have SacLink account, create one through mysalink.csus.edu.

Using the "CSC 133 – Communication Forum" you can send questions or discussion items regarding topics in CSC 133 to everyone in the class. This is a good way to ask your fellow students for clarifications about assignments, lecture topics, etc. Keep in mind that each such notice you send is read by everyone in the class (including the instructor). If you need to communicate privately with an instructor, use the instructor's individual email address as given above.

In addition to posting them to "CSC 133 – Communication Forum", the instructor may also send some of the important announcements directly to your SacLink e-mail.

Lateness/Absence/Drops:

It is very important for students to attend all classes and to be on time. Students who miss a class are responsible for all material discussed or handed out in the class.

Please inform yourself of the Department, College, and University policies on dropping courses. Policy information is available in the Computer Science Department office, Riverside Hall 3018.

Ethics:

When a student submits work to the instructor, it constitutes a contractual agreement that the work is solely that of the student. Further, submitted work carries with it an implicit agreement that the instructor may quiz the student in detail about the work.

Since the class is graded on a curve, if a student attempts to raise their grade through unethical means it is in essence causing a lowering of the grades of other students in the class. Further, any student who gives such help is equally guilty of unethical behavior for the same reasons. Therefore, all students are hereby notified that this course is being taught by instructors who will pursue attempts at cheating, since students who are cheating are cheating on you.

The **minimum penalty** for even a **single incident** of cheating in this course is **automatic failure of the course**; additional more severe penalties may also be applied. Note that cheating is grounds for dismissal from the University.

Please refer to the instructors' policy on academic integrity entitled "Ethics in Computer Science Classes", to the Computer Science Department's document entitled "Policy on Academic Integrity", and to the University's document entitled "Policy Manual section on Academic Honesty", which are all available online via the "Content" section of Canvas, for additional information. It is the responsibility of each student to be familiar with, and to comply with, the policies stated in these documents.

CSUS
COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 — Object-Oriented Graphics Programming

Name _____

SAMPLE EXAM

1. Write your name in the space above.
2. The exam is closed book, closed notes, except that you may use a *single sheet of hand-written notes*. If you use a note sheet you must put your name on it and turn it in with your exam.
3. There are 100 total points; you have 75 minutes to work on it — budget your time accordingly.
4. Absolutely NO use of ANY electronic devices is allowed during the exam. This includes cell phones, tablets, laptops, or any other communications device.
5. Please be neat — I cannot give credit to answers I cannot read.
6. The exam has 7 pages, counting this cover page. Make sure you have all the pages.

Problem	Points	Possible
1	_____	50
2	_____	15
3	_____	15
4	_____	20
Total	_____	100

1. Multiple Choice. Write the letter of the best answer in the blank to the left.

- _____ A certain Java/CN1 class named “**Point**” has constructors “**Point()**” and “**Point(int x, int y)**”. This is an example of
A. abstraction B. encapsulation C. inheritance D. overloading E. overriding
- _____ A certain Java/CN1 class named **Sphere** contains a method named **getColor()** which returns the color of the **Sphere** object. This method is an example of a (an)
A. accessor B. mutator C. aggregation D. design pattern E. abstraction
- _____ A certain Java/CN1 class named “B” extends another class named “A”. Class B defines a method named “C” with the same signature as that of a method named “C” in Class A. Method C does not contain the keyword “super”. A program constructs an instance of B and invokes method “C” in that object. The code which will be executed as a result of this invocation is
A. the code in A.C
B. the code in B.C
C. the code in A.C followed by the code in B.C
D. the code in B.C followed by the code in A.C
E. it depends on the code in A.C
F. it depends on the code in B.C
G. None of the above
- _____ If a Java/CN1 program contains a declaration such as “class A {...}”, where “...” represents the code defining the class, then
A. A has no parent class
B. A is its own parent
C. A is a superclass of Object
D. A is a subclass of Object
E. A is an abstraction of Object
- _____ In Java/CN1, *inheritance* is indicated using the keyword
A. abstract
B. extends
C. implements
D. static
E. new
F. none of the above

- _____ Before Java 8, an *interface* consists of
- A. a set of method declarations (abstract methods)
 - B. a set of method definitions (implementations)
 - C. a class description given in an online Application Programming Interface (API)
 - D. the set of classes in an inheritance hierarchy
 - E. a set of accessor (selector and/or mutator) methods
- _____ In a UML Class Diagram depicting classes named “Student” and “Course”, a label named “takes” on the diagram would most likely represent
- A. a method in Student
 - B. a method in Course
 - C. an association
 - D. a multiplicity
 - E. a composition
- _____ In CN1, when one object is registered as containing the method(s) to be invoked when another object generates an “ActionEvent”, we say the first object is a (an)
- A. event generator
 - B. action performer
 - C. listener
 - D. layout manager
 - E. exception handler
- _____ An association between two objects named “A” and “B” such that (1) B is referenced by A but not by any other object, and (2) the lifetime of B is controlled by A, is called a (an)
- A. Composition
 - B. Aggregation
 - C. Abstraction
 - D. Encapsulation
 - E. Inheritance
- _____ A CN1 build-in class *Container* is a
- A. component
 - B. layout manager
 - C. design pattern
 - D. framework
 - E. more than one of the above
 - F. none of the above

[THERE WOULD BE MORE MULTIPLE-CHOICE QUESTIONS IN THE REAL EXAM...]

2. Two different programming teams have implemented a class named **Rectangle**. One team provided accessors to get and set the location (origin), width, and height of a rectangle, while the other team chose to make the origin, width, and height fields public so that they can simply be directly accessed (read and/or changed). The second team argues that if you have accessors which allow you to both get and set all the values in the rectangle, there is no difference in having the fields public. Explain why the second team does not know what they are talking about. Be specific; give an example of how their approach can produce a software system that fails.

3. An CN1 program displays a form which contains two components: a control container containing a single button, and a separate display container. The button has an action listener attached to it which is an instance of a separate class. The display container has a pointer listener attached to it which is likewise an instance of a (different) separate class. The form is a subclass of **Form**; the containers are subclasses of **Container**, and the button is a **Button**.

Draw a UML diagram depicting the associations between the elements of this program.

4. Writing a CN1 code, describe on the next page the structure of a program which implements a simple Graphical User Interface (GUI) consisting of a form with a border layout which has the following components:

- (1) a single button on the north area, whose label is initially “Hello” and which prints the message “World” on the console when pressed;
- (2) a side menu with a single item which when selected has the effect of changing the button label to “Goodbye”;
- (3) a single container on the center area, on which pressing a pointer causes the background to become red, and releasing the pointer causes the background to become blue.

(note: CN1 **Button** class has a method `setText(String)` to change its label).

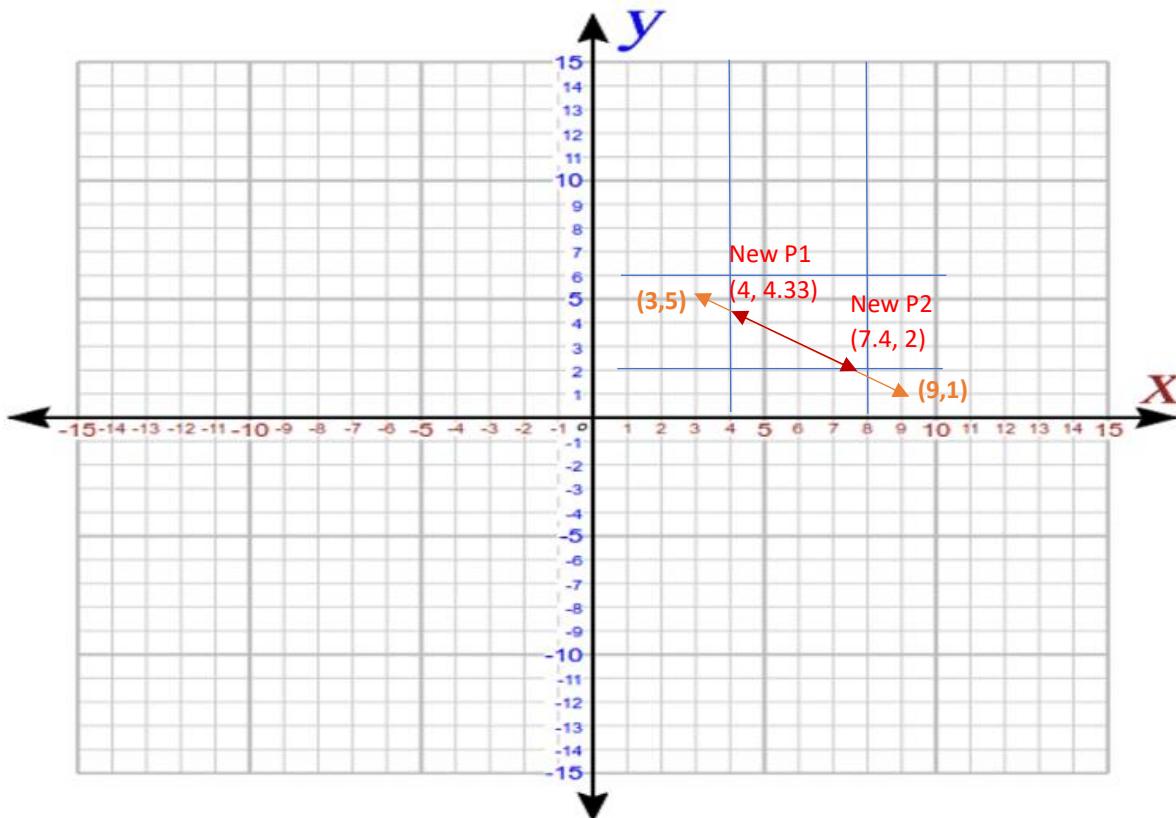
< use the NEXT PAGE for your answer to this question >

< This page is provided for your answer to the previous question >

Attendance Quiz # 10
Cohen-Sutherland Clipping Algorithm

- A clipping window has the following property:
 - $\text{Window}(\text{left}, \text{right}, \text{bottom}, \text{top}) = (4, 8, 2, 6)$
 - A Line, L1, with the following end points is drawn in the word: P1 (9,1), P2 (3,5)

Show how the Cohen-Sutherland clipping algorithm will clip this line and what its final endpoints are? Please show your work. Please draw the resultant line in the graph below.



P1 (9,1) ; P2(3,5)

$$M = (5 - 1) / (3 - 9)$$

$$M = 4/-6$$

$$M = -2/3$$

$$\begin{aligned} \text{New P1y} &= 1 + (-2/3)(4 - 9) \\ &= 1 + (-2/3)(-5) \end{aligned}$$

$$\text{New P1y} = 4.33$$

New P1 (4, 4.33)

$$\begin{aligned} \text{New P2y} &= 5 + (-2/3)(8 - 3) \\ &= 5 + (-2/3)(5) \end{aligned}$$

$$\text{New P2y} = 1.6$$

New P2 (8, -1.6)

Clip P2 again:

$$\text{New P2x} = 8 + (2 - (1.6)) / (-2/3)$$

$$= 8 + (0.4) / (-2/3)$$

$$\text{New P2x} = 7.4$$

New P2 (7.4, 2)

COHEN-SUTHERLAND ALGORITHM

PERSPECTIVE:

$$C1 = \text{code}(p1)=1000$$

$$C2 = \text{code}(p2)=0011$$

$$(C1 \& C2)=0000 - \text{Not Trivial Rejection}$$

P1 is not inside the window, so no swapping

Compute New P1 (4, 4.33)

New P1 is now inside the window, so swap P1 and P2

Compute New P2(which was swapped with P1) New P2 (8, -1.6)

P2 is still outside the window, so we Clip P2 again

Compute New P2(which was swapped with P1) New P2 (7.4, 2)

Now both C1 and C2 are 0000

Appendices

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_20\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_20
```

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) ;
    }
}
```

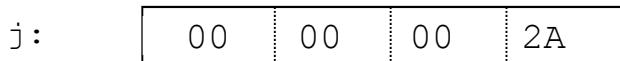
REFERENCES:

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

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

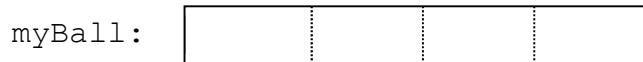


```
j = 42 ;           //assignment stores a value in the space:
```



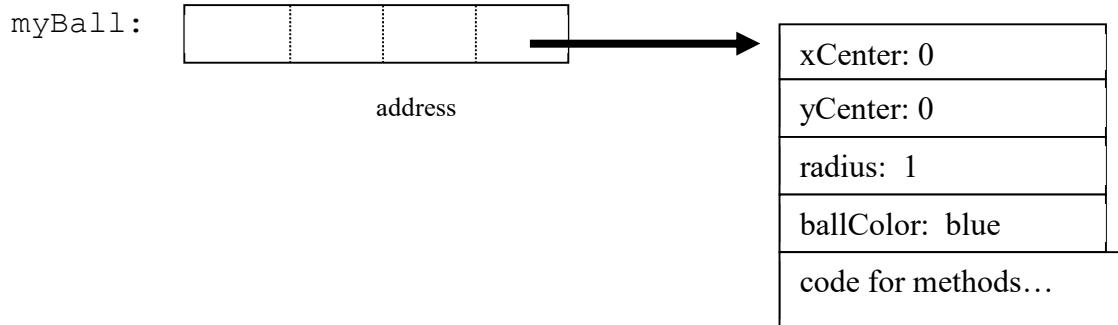
- A variable representing an object holds the address of the object, called a reference:

```
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 "equalsIgnoreCase()

- A common mistake:

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

This will not print the message

- Correct approach:

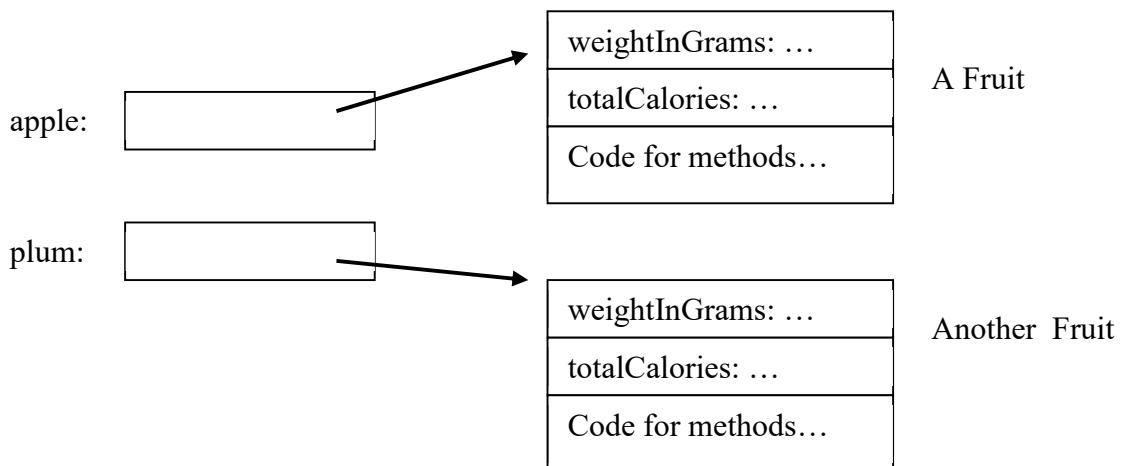
```
if (s1.equals("Ed")) { ... }
```

GARBAGE COLLECTION:

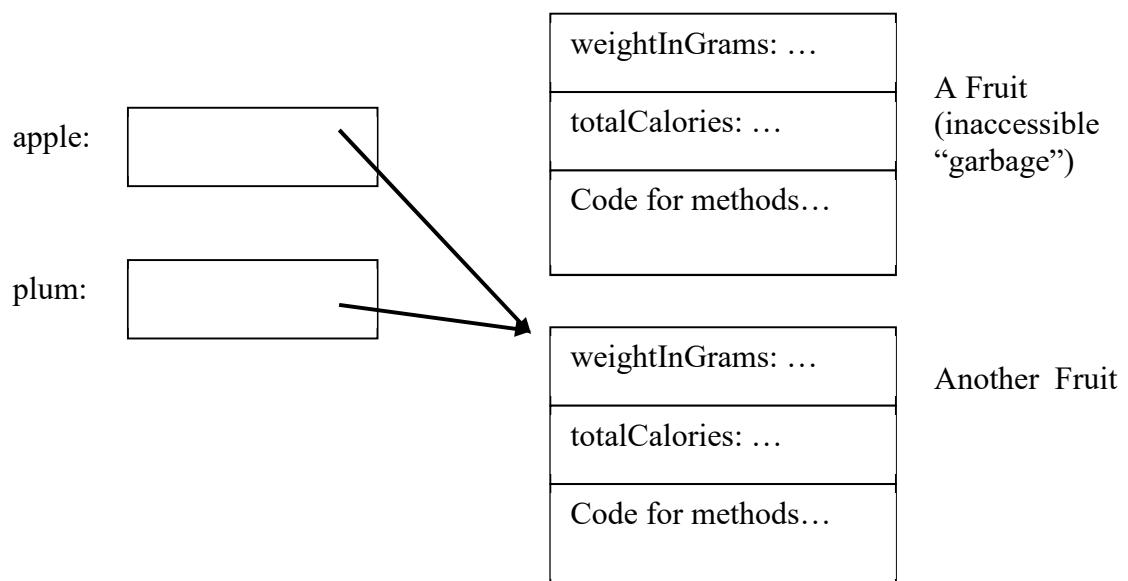
- Important characteristic: assignment does not copy objects; it copies references:

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

Before assignment:



After assignment:



ARRAYS:

Declaration:

```
int [ ] vals ;           // or: int vals [ ] ;  
int [ ] scores1, scores2 ; // two arrays of integers  
char [ ] grades, letters ; // two arrays of chars  
Point [ ] myPoints ; // array of objects
```

Characteristics:

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

```
int [ ] vals = new int [size] ; //size = # elements  
  
char grades [ ] = new char [5] ; // five elements  
  
Point [ ] myPoints = new Point [myScreen.getSize()] ;  
  
String [ ] words = new String [25]; // holds 25 String refs
```

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

ARRAYS, cont:

Common Declaration mistakes:

```
int [size] vals ;           //ILLEGAL - need 'new'  
int vals [size];           //ALSO ILLEGAL
```

Indexing:

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

```
vals [5] = 99 ;           // store primitives  
grades [3] = 'D' ;  
myPoints [i] = new Point (4,3) ; // store an object
```

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

Arrays as Objects:

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

```
vals.length // field (not method) giving array size;  
// length is "final" - cannot be changed
```

INITIALIZERS:

```
int [ ] vals = { 1, 3, 17, 99 } ;  
char [ ] letterGrades = { 'A', 'B', 'C', 'D', 'F' } ;
```

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

```
int [ ] vals = { 1, 3 } ;      // OK  
.  
. . .  
vals = { 1, 4, 7 } ;          // ILLEGAL (in any form)
```

ARRAYS and REFERENCES:

Potential Confusion:

```
int [ ] myInts = new int [10] ;  
  
System.out.println (myInts[4]) ;           // prints '0'  
  
.  
.  
.  
myInts[4] = 1776 ;  
  
System.out.println (myInts[4]) ;           // prints '1776'
```

but:

```
Ball [ ] myCollection = new Ball [10] ;  
  
System.out.println (myCollection[4]) ;       // runtime  
error!  
  
.  
.  
.  
myCollection [4] = new Ball (Color.blue) ;  
  
System.out.println (myCollection[4]) ;  
  
// prints string rep of Ball (if rep exists; else error)
```

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

ARRAYS and REFERENCES, cont:

Another easy “reference” ‘slip-up’:

```
int [ ] a = { 1, 2, 3 } ;  
  
int [ ] b = { 1, 2, 3 } ; //identical data  
  
. . .  
  
if ( a == b ) {  
  
    System.out.println ("arrays 'a' and 'b' are equal");  
  
}
```

- The above code does NOT print the message....
- Solution: “**java.util.Arrays**” [JDK 1.2 (and up)]
 - Contains methods which operate on arrays
 - Method **equals()** does an element-by-element comparison:

```
if ( Arrays.equals(a,b) ) {  
  
    System.out.println ("a and b are equal") ;  
  
}
```
 - Uses “**==**” for testing primitives
 - Uses (expects) a **.equals()** method to be defined for objects in arrays

ARRAYS OF ARRAYS:

Declaration :

```
int [ ] [ ] intTable ;      // "2D array" of ints  
Point [ ] [ ] pointTable ; // "2D array" of Points
```

Instantiation :

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

Result:

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

Accessing:

```
intTable [0] [2] = 5 ;           // assigns a primitive  
pointTable [0] [2] = new Point (17,-6) ; // assigns an object  
  
for (int i=0; i<3; i++) {  
    for (int j=0; j<5; j++) {  
        intTable [i] [j] = i * j ;  
        pointTable [i] [j] = new Point (i, j) ;  
    }  
}
```

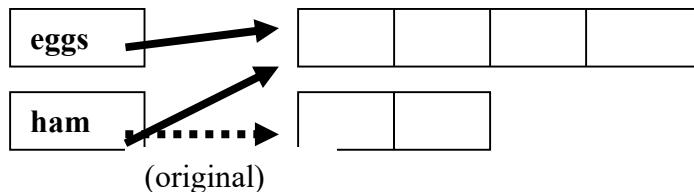
ARRAY ASSIGNMENT:

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

- Same element type
- Same number of dimensions:

```
int [ ] eggs = { 1, 2, 3, 4 } ;  
  
int [ ] ham = { 1, 2 } ;  
  
...  
  
ham = eggs ; //legal - same element type (int) and dimension  
  
ham [3] = 0 ; //legal - ham now has 4 elements!
```

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

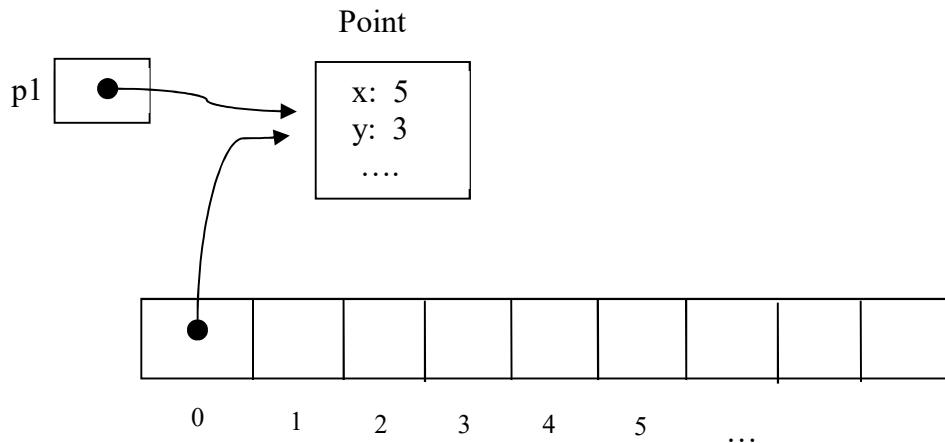


- Method `System.arraycopy()` can be used to perform a real “copy”

Vectors and ArrayLists :

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

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



- A common mistake:

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

Vectors and ArrayLists (cont.) :

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

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

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

Vectors and ArrayLists (cont.) :

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

```
add ( ) ;  
addElement ( ) ;  
clear ( ) ;  
capacity ( ) ;  
elementAt ( ) ;  
equals ( ) ;  
indexOf ( ) ;  
insertElementAt ( ) ;  
isEmpty ( ) ;  
removeElementAt ( ) ;  
size ( ) ;  
. . .
```

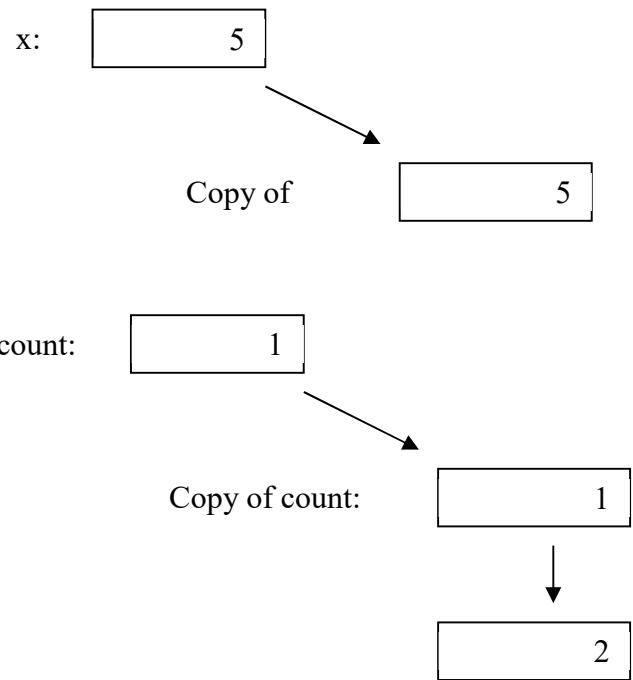
- How do you know what methods exist? And their parameters? And how they work?
- Answer: <https://docs.oracle.com/javase/8/docs/api/>
 - Complete online Java Standard Edition 8 API documentation
 - Can be downloaded to your machine

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 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

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:

$$\begin{array}{c} & \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \left(\begin{array}{ccc} X & X & X \\ X & X & X \\ X & X & X \\ X & X & X \end{array} \right) \end{array}$$

We characterize a matrix by giving the number of rows, then columns: the example above is a 4×3 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:

$$2 * \begin{array}{c} M1 \\ \left(\begin{array}{cc} 1 & 5 \\ -2 & 4 \\ 3 & 1 \end{array} \right) \end{array} = \begin{array}{c} M2 \\ \left(\begin{array}{cc} 2 & 10 \\ -4 & 8 \\ 6 & 2 \end{array} \right) \end{array}$$

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:

$$\begin{array}{c} \text{A} \\ \left(\begin{array}{cc} 1 & 5 \\ -2 & 4 \\ 3 & 1 \end{array} \right) \end{array} + \begin{array}{c} \text{B} \\ \left(\begin{array}{cc} 2 & 2 \\ -4 & 6 \\ -3 & 1 \end{array} \right) \end{array} = \begin{array}{c} \text{C} \\ \left(\begin{array}{cc} 3 & 7 \\ -6 & 10 \\ 0 & 2 \end{array} \right) \end{array}$$

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):

$$\left[\begin{array}{cc} 2 & 3 \end{array} \right] \quad \left[\begin{array}{ccc} 4 & 6 & -1 \end{array} \right]$$

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 columns in the matrix, 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 matrix.

$$\begin{array}{c} \text{V1} \\ \left[\begin{array}{cc} 2 & 3 \end{array} \right] \end{array} * \begin{array}{c} \text{M1} \\ \left[\begin{array}{ccc} 1 & 2 & -1 \\ 4 & 5 & 0 \end{array} \right] \end{array} = \begin{array}{c} \text{V2} \\ \left[\begin{array}{ccc} 14 & 19 & -2 \end{array} \right] \end{array}$$

It is important to note that the result of multiplying a vector by a matrix is always a new *vector*, and that the new vector always has the same number of elements (columns) as the original *matrix*. The elements of V2 above were formed by computing each of the following inner products (“•” represents the inner product or “dot product” operation) :

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

$$[2 \ 3] \cdot [2 \ 5] = (2*2) + (3*5) = 19 ; \text{ and}$$

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

5. Row-major vs. Column-major Notation

The previous section describes a vector as a matrix with a single *row*; however, a vector can also be viewed as a matrix with a single *column*. In this case the vector is written vertically, with the elements forming a column – for example:

$$\begin{bmatrix} 2 \\ 3 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 4 \\ 6 \\ -1 \end{bmatrix}$$

The difference in the two representations is simply one of convenience; the column form (also called “column-major form”) of a vector does not somehow represent a “different” vector than the equivalent row-major form.

However, along with the difference in notation comes a difference in representation of the multiplication operation. The multiplication of a (row) vector by a matrix always proceeds “from the left” – that is, the vector is always written on the left side of the multiplication sign (as shown in the preceding section), and the inner products are formed between the vector and consecutive *columns* of the matrix. When a vector is represented in “column-major” form, multiplication is always written with the vector on the *right* side, and the multiplication always proceeds from *right to left*.

For column-major representation, the elements of the result vector are formed by computing the inner products of the vector with each *row* of the matrix. This therefore means that for column-major representation (multiplication from the right), the number of elements (rows) of the vector must equal the number of *columns* in the matrix (as opposed to being equal to the number of rows in the matrix, which is the rule for row-major representation and multiplication from the left).

The form for column-major representation and multiplication from the right is shown below. Note that in this example the initial vector is written on the *right*, and the number of *columns* in the matrix matches the number of elements (rows) in the initial vector. Note also that the result (which is a vector, as before) is formed by computing the inner product of the initial vector with each *row* of the matrix (instead of with each column of the matrix as is done with row-major representation and multiplication

$$\begin{matrix} V2 \\ \left[\begin{array}{c} 14 \\ 19 \\ -2 \end{array} \right] \end{matrix} = \begin{matrix} M1 \\ \left[\begin{array}{cc} 1 & 4 \\ 2 & 5 \\ -1 & 0 \end{array} \right] \end{matrix} * \begin{matrix} V1 \\ \left[\begin{array}{c} 2 \\ 3 \end{array} \right] \end{matrix}$$

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^T . The following shows an example of a matrix A and its transpose A^T .

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

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:

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

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 $m \times p$ matrix (i.e, has “ m ” rows and “ p ” columns), and B is a $p \times n$ matrix (“ p ” rows and “ n ” columns), then the product $A * B$ produces a new matrix C which is $m \times n$. 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 2×3 matrix A by the 3×4 matrix B produces the 2×4 matrix C as shown:

$$\begin{array}{ccc|c} A & * & B & = & C \\ \left(\begin{matrix} 2 & 3 & -1 \\ 4 & 0 & 6 \end{matrix} \right) * \left(\begin{matrix} 1 & -1 & 0 & 4 \\ 2 & -2 & 1 & 3 \\ 5 & 7 & 1 & -3 \end{matrix} \right) & = & \left(\begin{matrix} 3 & -15 & 2 & 20 \\ 34 & 38 & 6 & -2 \end{matrix} \right) \end{array}$$

Each element of C in the above example is formed by computing the inner product of a row of A with a column of B. For example, the element with value “3” in row 0, column 0 of C is formed by the inner product of row 0 of A with column 0 of B: $(2*1) + (3*2) + (-1*5) = 3$. Likewise, the element with value “6” in row 1 (the second row), column 2 (the third column) of C is formed by the inner product of row 1 of A with column 2 of B: $(4*0) + (0*1) + (6*1) = 6$. Each of the other elements of C is likewise formed by computing the inner product of a row of A with a column of B.

Note that because the matrix multiplication operation is only defined when the number of columns of the first (left) matrix equals the number of rows of the second (right) matrix, in general it is *not* true that just because $A*B$ is a defined operation then it follows that $B*A$ is also well defined. In the example above, for instance, $A*B$ is defined (and produces the matrix C as shown), but the operation $B*A$ is *not defined* – because B does not have the same number of columns as the number of rows in A.

The only time that both $A*B$ and $B*A$ are well-defined matrix multiplication operations is when both A and B are *square* matrices of the *same size*.

Note that just because A and B are square matrices of the same size (and hence both $A*B$ and $B*A$ are well-defined), it is not in general true that $A*B = B*A$. That is, *the matrix multiplication operation is not commutative*. This property (lack of commutativity of matrix multiplication) is important to keep in mind when manipulating matrices.

8. Identity Matrix

We define a special matrix form called the *Identity matrix*. The Identity matrix has the properties that (1) it is square; (2) it has 1’s in every element along its major diagonal; and (3) it has zeroes in every element not on its major diagonal. For example, the following shows an Identity matrix I of size 3:

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Identity matrices have certain important and useful properties. For example, if I is an identity matrix of size n , and A is also a square matrix of size n , then it is the case that

$$A * I = I * A = A.$$

That is, multiplying matrix A by the identity matrix, whether on the left or on the right, leaves A unchanged (this is in fact the mathematical definition of an “identity element”; for example, the value “1” is the identity element with respect to the ordinary algebra operation “multiplication” – multiplying a given number by 1 does not change its value).

9. Matrix Inverses

A given matrix “M” is said to be *invertible* (or “*has an inverse*”), if there exists another matrix, denoted as M^{-1} , such that

$$M * M^{-1} = M^{-1} * M = I,$$

where I is the Identity matrix of the same size as M. The following general properties relate to matrix inverses:

- (1) only *square* matrices have inverses (this is a consequence of the above definition, which requires the same result for multiplication from the left and right);
- (2) not all square matrices have inverses (that is, there are some matrices for which, even though they are square, it is not possible to find another matrix for which the above equations hold); and
- (3) if a matrix has an inverse, it will be *unique* (that is, every matrix has at most one inverse).

An important observation about matrix inverses is that they represent, in a sense, an “opposite” operation. That is, if M represents some operation (say, a transformation of some kind), then M^{-1} represents the “opposite transformation”. This observation is useful when attempting to build a series of transformations which are represented by matrices; it provides a method of specifying how to “undo” a given operation. However, care must be taken to insure that the operation in question (being represented by a matrix) is “undoable” (i.e. that the matrix is invertible). See the section on “Inverse Of Products” for further information on this topic.

10. Associativity and Commutativity

As noted above, the matrix multiplication operation is *not* commutative. That is, it is *not* true (in the general case) that $A*B$ produces the same result (matrix) as $B*A$.

However, an important property of matrix multiplication is that it 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 P_1 . This is done by multiplying the point (represented as a vector as described above) by the matrix, which results in a new (translated) point P_2 . As noted above, matrix multiplication can be done either using “row-major” form or “column-major” form. However, most programming language matrix libraries (including those in Java) use column-major form – that is, matrix multiplication is done “from the right”. Assuming we are using this convention (for example, assuming we are writing in Java), then when you multiply a point by a transformation matrix, you do so “from the right”, meaning that the point is written on the right hand side with the matrix to its left, and the multiplication proceeds that way (“from right to left”).

The algebraic representation of the above example would be the following, where “P1” is the original point and “M” is the matrix containing the desired translation:

$$\begin{pmatrix} P_2 \\ X' \\ Y' \end{pmatrix} = \begin{pmatrix} M \\ \quad \quad \quad \end{pmatrix} * \begin{pmatrix} X \\ Y \end{pmatrix}$$

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.

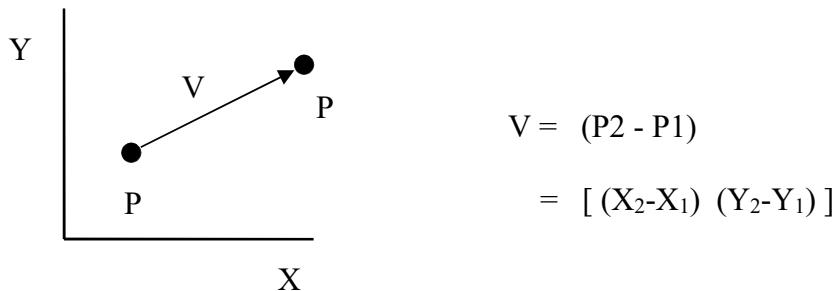
Therefore, if you have a certain order in which you want a set of transformations *applied*, you must get them into the transformation matrix such that they are in order, from *right* to *left*, in the order that you want them applied. If you have, say, a scale and a translate to be applied, and you 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.

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.

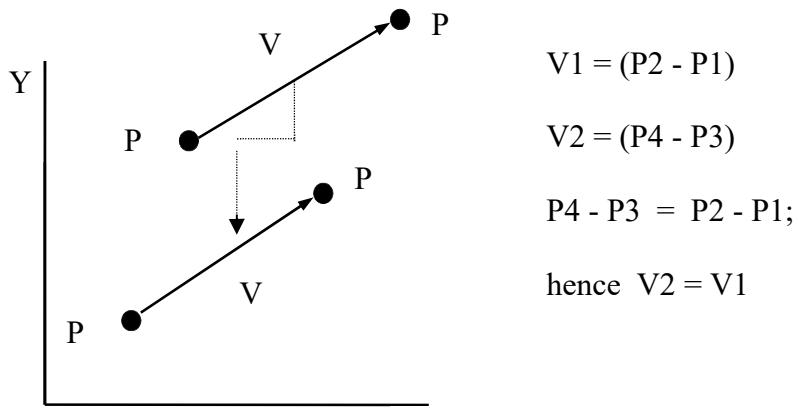
A vector runs from one point to another, and can be represented as the *difference* between the two points. The “difference between two points” is obtained by taking the difference between corresponding elements of the two points. Thus, in 2D:



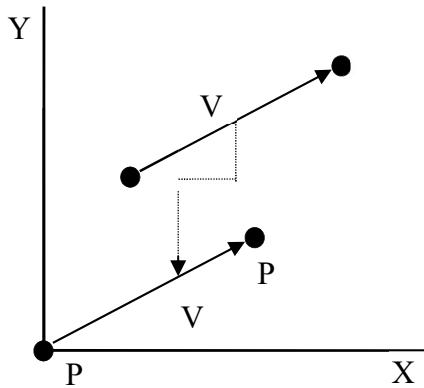
Note that a vector is represented as an object in square brackets; the elements inside the square brackets are called the *components* of the vector. Each component of a vector is a numerical quantity.

2. Translation

Vectors can be *translated* without altering their value (since they consist of *magnitude* and *direction* but not *position*):



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:

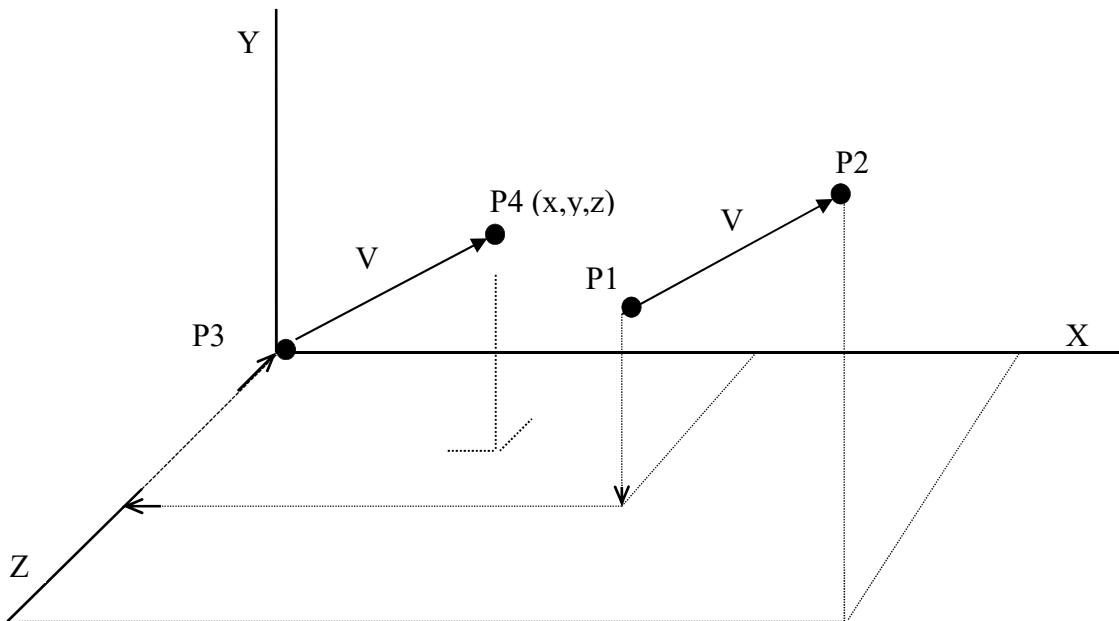


$$\begin{aligned}
 \mathbf{V} &= (\mathbf{P}_2 - \mathbf{P}_1) \\
 &= [(X_2 - X_1) \ (Y_2 - Y_1)] \\
 &= [(X_2 - 0) \ (Y_2 - 0)] \\
 &= [(X_2) \ (Y_2)] \\
 &= [X_2 \ Y_2]
 \end{aligned}$$

Thus, a vector can also be represented as a *single point*: the point at the head of the vector when the tail is at the origin. Both representations of vectors (as the difference of two points or as a single point) are useful.

3. Two-Dimensional vs. Three-Dimensional Vectors

Vectors in 3D work the same way as in 2D. They can be specified as the difference between two (3D) points; they can be translated without changing their value; and they can be specified as a single (3D) point since the tail can be translated to the origin. (In fact, a 2D vector can be considered to be a special case of a 3D vector, with the Z component equal to zero.)



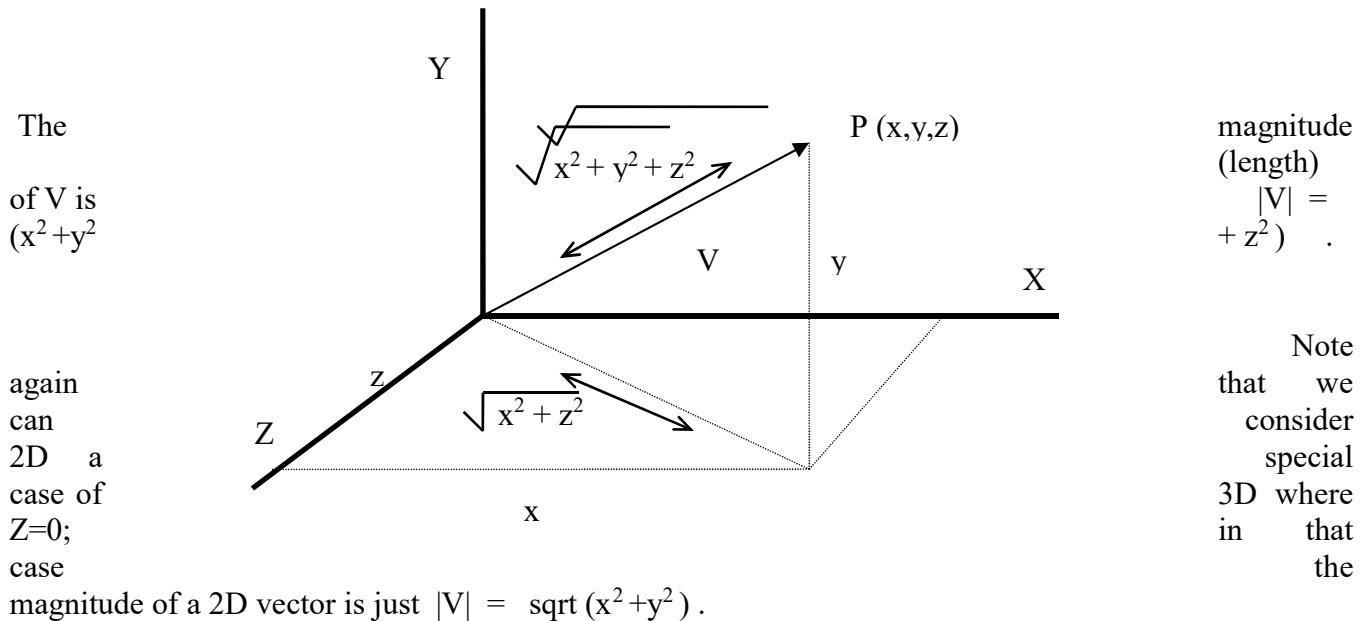
In the preceding figure,

$$\begin{aligned}
 V &= (P_2 - P_1) \\
 &= (P_4 - P_3) \\
 &= [(X_2 - X_1) (Y_2 - Y_1) (Z_2 - Z_1)] \\
 &= [(X_4 - X_3) (Y_4 - Y_3) (Z_4 - Z_3)] \\
 &= [(X_4 - 0) (Y_4 - 0) (Z_4 - 0)] \\
 &= [X_4 \ Y_4 \ Z_4] \\
 &= [x \ y \ z]
 \end{aligned}$$

Note therefore that a single point (x, y, z) in 3 dimensions represents the 3D vector $[x \ y \ z]$, the vector from the origin to (x, y, z) .

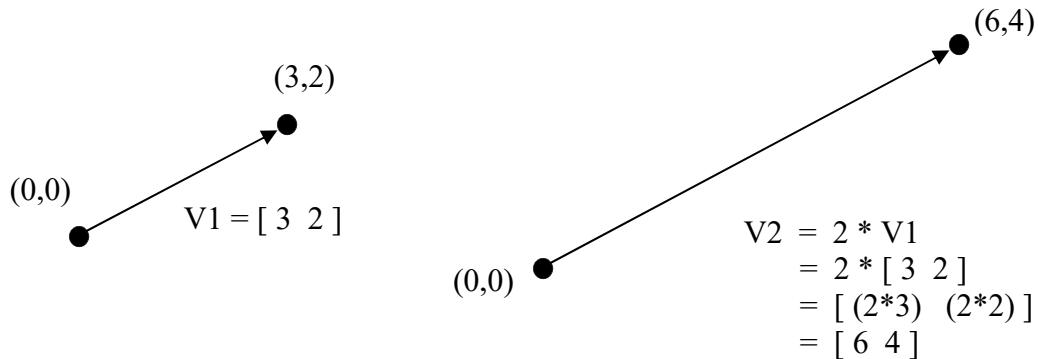
4. Magnitude

The *length* or *magnitude* of a vector V is denoted by the symbol $|V|$, and can be calculated using the Pythagorean Theorem:



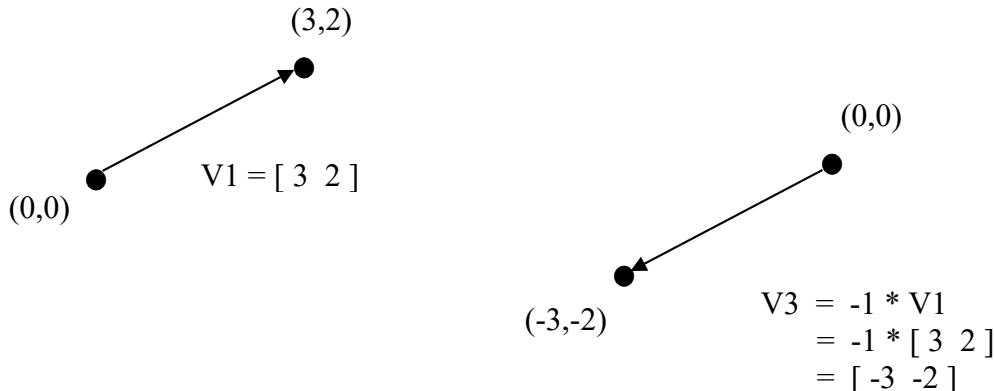
5. Scalar Multiplication

Multiplying a vector by a scalar value is a well-defined operation, and produces a new vector whose components are the result of multiplying each of the original vector components by the scalar value. For example, multiplying the 2D vector $V1 = [3 \ 2]$ by the scalar value 2 produces the new vector $V2 = [6 \ 4]$. This is shown graphically in 2D as:



Note that the effect of multiplying a vector by a positive scalar value is to produce a new vector whose direction is the same as the original vector and whose magnitude varies according to the scalar value: multiplying by a value greater than one increases the magnitude, while multiplying by a fractional value decreases the magnitude.

Note also that multiplying a vector by a *negative* scalar value has the effect of reversing the direction of the vector. For example, using the vector $V1 = [3 \ 2]$ (as shown above), a new vector $V3 = -1 * V1$ is $[-3 \ -2]$, as shown below (keep in mind that vectors can be translated without changing their value):



6. Unit Vectors

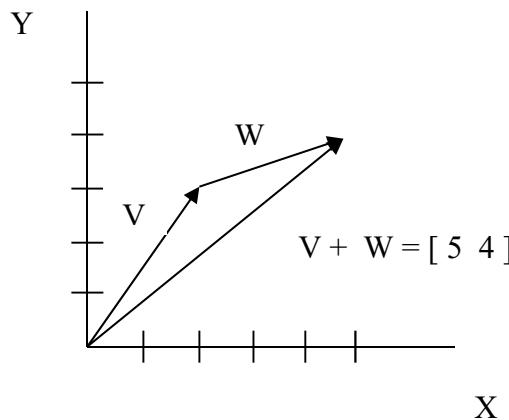
Given a vector V , there exists a corresponding vector U (also sometimes denoted as V with a “^” over it) called a *unit vector* with the property than U has the same *direction* as V but has *length* = 1.

The unit vector U for a given vector V is calculated by dividing each of the components of V by the magnitude (length) of V : $U = V / |V| = [(V_x / |V|) \ (V_y / |V|)]$. For example, if $V = [3 \ 4]$, then $|V| = \sqrt{3^2 + 4^2} = 5$, and $U = [(3/5) \ (4/5)] = [0.6 \ 0.8]$. Note that this is a vector in the same direction as V , and whose length is $|U| = \sqrt{0.6^2 + 0.8^2} = \sqrt{0.36 + 0.64} = \sqrt{1.0} = 1.0$.

7. Vector Addition

Given two vectors V and W , the vector sum $V+W$ is a well-defined operation and produces a new vector whose components are the sums of the corresponding components of V and W . For example, if $V = [2 \ 3]$ and $W = [3 \ 1]$, then the vector $V+W = [(2+3) \ (3+1)] = [5 \ 4]$.

Vector addition can be represented graphically by placing the tail of one vector at the head of the other; the sum will be the vector from the tail of the first vector to the head of the second. For example, using the vectors V and W given above:



8. Dot Product

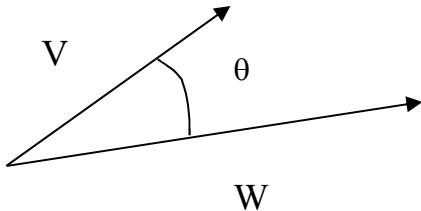
Another well-defined operation on vectors is called the *dot product* (also called the *inner product*). Given two vectors V and W , the dot product is written notationally as $V \bullet W$ and is defined as a scalar value $D = \sum (V_i * W_i)$ over all components “ i ” of V and W . Note that the dot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

$$V \bullet W = ((2 * 7) + (3 * -1)) = (14 - 3) = 11. \quad \text{As noted, this result is a scalar value.}$$

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 ,

$$V \bullet W = |V| * |W| * \cos(\theta), \text{ where } \theta \text{ is the angle between } V \text{ and } W:$$



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

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

Assignment#0

- Find a lab computer that has CN1 or install CN1 to your computer.
- Following the instruction in the lecture slides, generate an empty project called A0Prj.
- Modify Starter.java by replacing the texts “Hi World” with “**Assignment#0 –Spring 2019**”. Run the simulator.
- Experiments with debugging options of your IDE.
- Verify that your submission also works using the command line and **RunAssignment.jar** program.

Do submit A0 via Canvas for Grading (its purpose is to make sure you have access to CN1 and ready to solve real assignments)

Assignment#0 Deliverables

- Turn in YourLastName-YourFirstName-a0.zip file to Canvas for grading. This zip file shall consist of:
A0Prj.jar (under *dist* dir) and entire *src* dir
- Turn in a separate PDF document with the following:
 - A screen capture of your program's execution (see the Iphone image showed in lecture)
 - A screen capture of a Windows where a **command line(s)** was used to launch the application. (see slide # 19 and 21 (do both) of 2 – Introduction to Mobile App Development and CN1 lecture)
 - A screen capture of a breakpoint (3) of the following lines (in your IDE debugger) – showing during execution:
 - *updateNetworkThreadCount(2)* – In **init**(Object context)
 - *if(current != null){* – In **start()**
 - *current = getCurrentForm();* – In **stop()**

Assignment #1: Class Associations & Interfaces

Due Dates:

1st Delivery: 4 Test cases – February 11th, 2019 [1 week]

2nd Delivery: UML Class Diagram + Source Files + Jar file - February 25th, 2019 [2+ weeks]

Introduction

In this Spring 2019 semester, we will be studying object-oriented computer graphics programming by developing a program which is a variation of a classic arcade and home video game called “Asteroids” (Version 5 – 8/4/18). In this game, you will be controlling a space ship flying through a field of asteroids, firing missiles to destroy asteroids which threaten to collide with and destroy your ship. Occasionally, enemy ships attempt to impede progress. Various other graphical operations will also be implemented to extend your game beyond the original version.

If you have never seen the original Asteroids game, you can see a sample of the (Arcade) Asteroids game in Youtube. <https://www.youtube.com/watch?v=WYSupJ5r2zo>. However, your game will be slightly different, and it is not necessary to be familiar with the original Asteroids game to do any of the assignments during the semester.

The initial version of your game will be *text-based*. As the semester progresses we will add graphics, animation, and sound. The goal of this first assignment is to develop a good initial class hierarchy and control structure, and to implement it in CN1/Java. This version uses keyboard input commands to control and display the contents of a “game world” containing a set of objects in the game. In future assignments, many of the keyboard commands will be replaced by interactive GUI operations or be replaced with animation functions, but for now we will simply simulate the game in “text mode” with user input coming from the keyboard and “output” being lines of text on the screen.

Program Structure

Because you will be working on the same project all semester, it is extremely important to organize it correctly from the beginning. Pay careful attention to the class structure described below and make sure your program follows this structure accurately.

The primary class encapsulates the notion of a **Game**. A game in turn contains several components: (1) a **GameWorld** which holds a collection of *game objects* and other state variables, and (2) a **play()** method to accept and execute user commands. Later, we will learn that a component such as GameWorld that holds the program’s data is often called a **model**.

The top-level *Game* class also encapsulates the *flow of control* in the game (such a class is therefore sometimes called a **controller**). The controller enforces rules such as what actions a player may take and what happens as a result. This class accepts input in the form of keyboard commands from the human player and invokes appropriate methods in the game world to perform the requested commands – that is, to manipulate data in the game model.

In this first version of the program the top-level game class will also be responsible for displaying information about the state of the game. In future assignments, we will learn about a separate kind of component called a **View** which will assume that responsibility.

When you create your CN1 project, you should name the main class as **Starter**. Then you should modify the start() method of the Starter class so that it would construct an instance of the Game class. The other methods in Starter (i.e., init(), stop(), destroy()) should not be altered or deleted. The Game class must extend from the build-in Form class (which lives in **com.codename1.ui** package). The Game constructor instantiates a GameWorld, calls a GameWorld method init() to set the initial state of the game, and then starts the game by calling a Game method play(). The play() method then accepts keyboard commands from the player and invokes appropriate methods in GameWorld to manipulate and display the data and game state values in the game model. Since CN1 does not support getting keyboard input from command prompt (i.e., the standard input stream, System.in, supported in Java is not available in CN1) the commands will be entered via a text field added to the form (the Game class). Refer to “Appendix – CN1 Notes” for the code that accepts keyboard commands through the text field located on the form.

The following shows the pseudo-code implied by the above description. It is important for things that we will do later that your program follows this organization:

```
class Starter {
    //other methods
    public void start() {
        if(current != null){
            current.show();
            return;
        }
        new Game();
    }
    //other methods
}
```

```
public class GameWorld {
    public void init(){
        //code here to create the
        //initial game objects/setup
    }
    // additional methods here to
    // manipulate world objects and
    // related game state data
}
```

```
import com.codename1.ui.Form;
public class Game extends Form{
    private GameWorld gw;

    public Game() {
        gw = new GameWorld();
        gw.init();
        play();
    }

    private void play() {
        // code here to accept and
        // execute user commands that
        // operate on the game world
        //(refer to "Appendix - CN1
        //Notes" for accepting
        //keyboard commands via a text
        //field located on the form)
    }
}
```

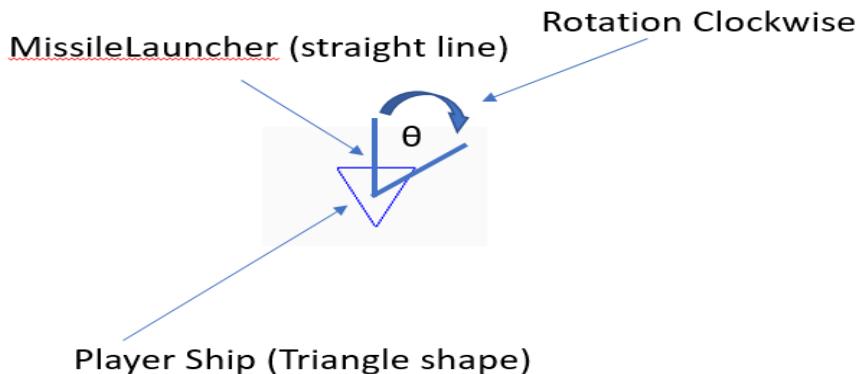
Game World Objects

For now, assume that the game world size is fixed and covers 1024(width) x 768(height) area (although we are going to change this later). The origin of the “world” (location (0,0)) is the lower left hand corner. The game world contains a collection which aggregates objects of abstract type **GameObject**. There are two kinds of abstract game objects: “fixed objects” with fixed locations (which are fixed in place) and “moveable objects” with changeable locations (which can move or be moved about the world). For this first version of the game there is just a single kind of fixed object: a **space station**; and there are three kinds of moveable objects: **ships (Player Ship (PS) and Non-Player Ship (NPS), missiles** (which are objects fired by ships), and **asteroids**. Later we will see that there are other kinds of game objects as well.

The various game objects have attributes (fields) and behaviors (“functions” or “methods”) as defined below. These definitions are requirements which must be properly implemented in your program.

- All game objects have a *location*, defined by floating point values X and Y. (you can use float or double to represent it) non-negative values X and Y which initially should be in the range 0.0 to 1024.0 and 0.0 to 768.0, respectively. The point (X,Y) is the center of the object. Hence, initial locations of all game objects should always be set to values such that the objects' centers are contained in the world. All game objects provide the ability for external code to obtain their location. By default, game objects provide the ability to have their location changed, unless it is explicitly stated that a certain type of game object has a location which cannot be changed once it is created. Except for a player **ship**, initial locations of all the game objects should be assigned randomly when created.
 - Note: For future usages (i.e. specifying a point in a space), you might consider encapsulating a pair of coordinate values (X,Y) using a CN1's built-in classes Point or Point2D.
See: www.codenameone.com/javadoc/com/codenname1/ui/geom/Point.html
[www.codenameone.com/javadoc/com/codenname1/ui/geom/Point2D.html](http://www.codenameone.com/javadoc/com/codename1/ui/geom/Point2D.html)
- All game objects have a color, defined by a int value (use static `rgb()` method of CN1's built-in **ColorUtil** class to generate colors). All objects of the same class have the same color (chosen by you), assigned when the object is created (e.g, ships could be green, missiles could be red, asteroids could be blue). All game objects provide the ability for external code to obtain their color. By default, game objects provide the ability to have their color changed, unless it is explicitly stated that a certain type of game object has a color which cannot be changed once it is created.
- Moveable objects have attributes *speed* and *direction (heading)*, which are used to define how they move through the world when told to do so. *Speed* is an integer in the range **0-15**, initialized to a random value in that range unless otherwise stated. *Direction* is an integer in the range 0..359, initialized to a **random value** in that range unless specified otherwise, indicating a compass heading. (Thus, a direction of zero specifies heading "north" or "up the screen"; a direction of 90 specifies heading "east" or toward the right side of the screen, etc.)
- Moveable objects provide an interface that allows other objects to tell them to move. Telling a moveable object to *move()* causes the object to update its location based on its current speed and direction (heading). See below for details on updating a movable object's position when its *move()* method is invoked.
- Some moveable objects are *steerable*, meaning that they provide an interface that allows other objects to tell them to change their directions.
- *Player Ship (PS)* is steerable objects whose speed and direction (heading) can be controlled by a player. There is only *one player ship* in the game at any one time, and its initial location should be the center of the world (e.g, 512,384), with a speed of zero and a heading of zero (north). PS has an attribute called *missile count*, which is an integer representing the number of missiles the ship has available to be fired. PS starts with the maximum number of missiles, which is 10. Each time a ship fires a missile its missile count is decremented by one; when missile count reaches zero it can no longer fire any missiles. However, ships that fly to a space station automatically have their missile count reloaded to the original maximum number.

- A PS owns a MissileLauncher. It is a steerable object whose speed and location is the same as a PS. However, a direction of a MissileLauncher can be different from a PS. There is only one MissileLauncher per PS. Its initial location should be at the center of the world with speed of zero and heading of north. Graphically, in future Assignment 3 (a3) – not in this assignment, a MissileLauncher can be appeared like this when initialized:



Both classes, PlayerShip and MissileLauncher are required to have a **composition** relation.

- Non-Player Ship(s) (NPS) is NOT steerable objects. NPS(s) are movable objects that fly through space at a fixed speed and direction. NPS(s) also have a fixed attribute *size* (*small 10 or large 20*), an integer giving the dimensions of the NPS(s). The size (small or large), speed, and direction of a NPS are to be randomly generated when it is created. If a player's missile collides with a NPS, the NPS (and the missile) are destroyed and removed from the game, and the user scores some points (you may decide how many). If a PS collides with a NPS, the PS and the NPS are destroyed and the player loses one “life”. A complete game consists of playing until three PS have been destroyed. A NPS also has a MissileLauncher. However, it is not steerable. It has a same direction a NPS. A NPS can randomly fires missile at PS (a maximum of 2). If a NPS’s missile collides with a PS, the player loses one life and a missile is removed from the game.
- Missiles are moveable objects that are fired from ships’s MissileLauncher. When a missile is fired, it has a direction matching its ship’s MissileLauncher (that is, missiles are fired out the front of a launcher), and a speed that is some fixed amount greater than the speed of its ship (you may decide the amount). Missiles also have an attribute called *fuel level*, a positive integer representing the number of time units left before the missile becomes dead and disappears. A missile’s fuel level gets decremented as time passes, and when it reaches zero the missile is removed from the game. All missiles have an initial fuel level of 10. Note that while missiles are *moveable*, they are not *steerable*; that is, there is no way to change a missile’s speed or heading once it is fired. The program must enforce this constraint.
- Asteroids are movable objects that tumble through space at a fixed speed and direction. Asteroids also have a fixed attribute *size* (*range 6..30 – Note: We will relax this assumption in later assignment*), an integer giving the dimensions of the asteroid. The size, speed, and direction of an asteroid are to be randomly generated when the asteroid is created. If a PS’s missile collides with an asteroid, the asteroid (and the missile) are

destroyed and removed from the game, and the user scores some points (you may decide how many). If a player *ship* collides with an asteroid, the ship and the asteroid are destroyed and the player loses one “life”. A complete game consists of playing until three player ships have been destroyed.

- Fixed objects have unique integer *identification (id) numbers*; every fixed object (of any subtype) has an integer id which is different from that of every other fixed object. (Note that this is an appropriate place to use a *static int* in a Java class; such a value can be used to keep track of the next available id number each time an instance of the class is constructed.) The program must enforce the rule that it is not possible to change either the ID number or the location of a fixed object once it has been created.
- *Space stations* are fixed objects that have an attribute called their *blink rate*; each space station blinks on and off at this rate (blink rate specifies the *period*; a blink rate of 4, for example, means the light on the space station is on for four seconds and then off for four seconds). Blink rate is initialized to a random value between zero and four. Space stations also house an unlimited supply of missiles; any ship arriving at a space station automatically gets a full resupply of missiles (up to the maximum number of missiles a ship can hold)

The preceding paragraphs imply several *associations* between classes: an inheritance hierarchy, interfaces such as for *steerable* and *moving* objects, and aggregation associations between objects and where they are held. **You must first develop a UML diagram for the relationships, and then implement it in a Java/CN1 program.** Appropriate use of encapsulation, inheritance, aggregation, abstract classes, and interfaces are important grading criteria.

You must use a tool to draw your UML (e.g., Violet or any other UML drawing tool) and output your UML as a **pdf file** (e.g., print your UML to a pdf file). Your UML must show all important associations between your entities and utilize correct graphical notations. For your entities, you must use three-box notation (as mentioned in lecture) and show all the important fields and methods. You must indicate the visibility modifiers of your fields and methods, but you are not required to show parameters in methods and return types of the methods.

Game Play

When your program begins, there are no objects in the world. The player can enter commands (see below) to control various aspects of the game such as adding objects, moving the ship, firing missiles, etc. The game keeps track of time using a clock, which is simply an integer counter. The objective is to achieve the highest possible score in the minimum amount of time – that is, to destroy as many asteroids and NPS(s) as possible, flying to a space station to restock missiles as necessary, without losing ships, all in minimum time.

Commands

Once the game world has been created and initialized, the Game constructor is to call a method name play() to actually begin the game. play() accepts single-character commands from the player via the text field located on the form (the Game class) as indicated in the “Appendix – C1 Notes”.

Any undefined or illegal input should generate an appropriate error message on the console and ignore the input. Single keystrokes invoke action -- the human hits “enter” after typing each command. **Additionally, any input condition where a function cannot be carried out, your program must output an error text. For example, user inputs a ‘d’ (decrease a**

speed of a player ship) command where a ‘s’ command (add a player ship to the world) was not initiated before. The allowable input commands and their meanings are defined below (note that commands are case sensitive):

‘**a**’ – add a new asteroid to the world.

‘**y**’ – add a NPS to the world.

‘**b**’ – add a new blinking space station to the world.

‘**s**’ – add a PS to the world.

‘**i**’ – increase the speed of the PS by a small amount.

‘**d**’ – decrease the speed of the PS by a small amount, as long as doing so doesn’t make the speed go negative (the ship can’t go backwards).

‘**l**’ (ell) – turn PS left by a small amount (change the heading of the ship by a small amount in the counter-clockwise direction). Note that determination of the actual new heading is not the responsibility of the game class; it simply instructs the game world that a change is to be made.

‘**r**’ – turn PS right (change the ship heading by a small amount in the clockwise direction).

‘**>**’ - a user controls the direction of the PS’s MissileLauncher by revolving it about the center of the player ship in a clockwise direction. This command turns the MissileLauncher by small θ angle. You decide a value.

‘**f**’ – fire a missile out the front of the PS. If the ship has no more missiles, an error message should be printed; otherwise, add to the world a new missile with a location, speed, and launcher’s heading determined by the ship.

‘**L**’ – Launch a missile out the front of the NPS. If the ship has no more missiles, an error message should be printed; otherwise, add to the world a new missile with a location, speed, and launcher’s heading determined by the ship.

‘**j**’ – jump through hyperspace. This command causes the ship to instantly jump to the initial default position in the middle of the screen, regardless of its current position. (This makes it easy to regain control of the ship after it has left visible space; later we will see a different mechanism for viewing the ship when it is outside screen space.)

‘**n**’ – load a new supply of missiles into the PS. This increases the ship’s missile supply to the maximum. It is permissible to reload missiles before all missiles have been fired, but it is not allowed for a ship to carry more than the maximum number of missiles. Note that normally the ship would actually have to fly to a space station to reload; for this version of the game we will assume the station has transporters that can transfer new missiles to the ship regardless of its location.

‘**k**’ – a PS’s missile has struck and killed an asteroid; tell the game world to remove a missile and an asteroid and to increment the player’s score by some amount of your choosing. You may choose any missile and asteroid to be removed; later we’ll worry about missiles needing to be “close to” their victims.

‘**e**’ – a PS’s missile has struck and eliminated a NPS; tell the game world to remove a missile and a NPS and to increment the player’s score by some amount of your choosing. You

may choose any missile and any NPS to be removed; later we'll worry about missiles needing to be "close to" their victims.

'E' – a NPS's missile has struck and Exploded a PS; tell the game world to remove a missile and a PS. You may choose any missile and to decrement the count of lives left; later we'll worry about missiles needing to be "close to" their victims.

'c' – the PS has crashed into an asteroid; tell the game world to remove the ship and an asteroid and to decrement the count of lives left; if no lives are left then the game is over. You may choose any asteroid to be removed; later we'll worry about asteroids needing to be "close to" the ship.

'h' – the PS has hit a NPS; tell the game world to remove the NPS and to decrement the count of lives left; if no lives are left then the game is over. You may choose any NPS to be removed; later we'll worry about NPS needing to be "close to" the PS.

'x' -- two asteroids have collided with and exterminated each other; tell the game world to remove two asteroids from the game. You may choose any two asteroids to be removed; later we'll worry about the asteroids needing to be "close to" each other. You may ignore collisions between NPSs.

'I' -- one asteroid have collided and impacted the NPS; tell the game world to remove them from the game. You may choose one asteroid and one NPS to be removed; later we'll worry about the asteroid needing to be "close to" NPS.

't' – tell the game world that the "game clock" has ticked. Each tick of the game clock has the following effects: (1) all moveable objects are told to update their positions according to their current direction and speed; (2) every missile's fuel level is reduced by one and any missiles which are now out of fuel are removed from the game; (3) each space station toggles its blinking light if the tick count modulo the station's blink rate is zero; and (4) the "elapsed game time" is incremented by one.

'p' – print a display (output lines of text) giving the current game state values, including: (1) current score (number of points the player has earned), (2) number of missiles currently in the ship, and (3) current elapsed time. Output should be well labeled in easy-to-read format.

'm' – print a "map" showing the current world state (see below).

'q' – quit, by calling the method `System.exit(0)` to terminate the program. Your program should first confirm the user's intent to quit before exiting.

Additional Details

- Each command must be encapsulated in a separate method in the Game class (later we will move those methods to other locations). Most of the game class command actions also invoke related actions in *GameWorld*. When the Game gets a command, it invokes one or more methods in the *GameWorld*, rather than itself manipulating game world objects.
- All classes must be designed and implemented following the guidelines discussed in class:
 - *All data fields must be private,*
 - *Accessors / mutators must be provided, but only where the design requires them,*
 - *Game world objects may only be manipulated by calling methods in the game world. It is never appropriate for the controller to directly manipulate game world data attributes.*

Moving objects need to determine their new location when their *move()* method is invoked, at each time tick ('t' command). The new location can be computed as follows:

```
newLocation(x,y) = oldLocation(x,y) + (deltaX, deltaY), where  
deltaX = cos(theta)*speed,  
deltaY = sin(theta)*speed, and  
theta = 90 - heading.
```

In this assignment, we are assuming "time" is fixed at one unit per "tick", so "elapsed time" is 1.

- For this assignment, all output will be in text on the console; no "graphical" output is required. The "map" (generated by the 'm' command) will simply be a set of lines describing the objects currently in the world, similar to the following:

```
Player Ship: loc=130.0,565.5 color=[0,255,0] speed=8 dir=90 missiles=5 Missile launcher dir = 50  
PS's Missile: loc=180.3,100.0 color=[255,0,0] speed=10 dir=112 fuel=7  
NPS's Missile: loc=50.9,540.2 color=[255,0,0] speed=5 dir=313 fuel=1  
Non-Player Ship: loc=640.2,20.0 color=[128,128,128] speed=6 dir=20 size=10  
Asteroid: loc=440.2,20.0 color=[128,128,128] speed=6 dir=0 size=6  
Asteroid: loc=660.0,50.0 color=[128,128,128] speed=1 dir=90 size=26  
Station: loc=210.2,800.0 color=[255,255,0] rate=4
```

Note that the appropriate mechanism for implementing this output is to override the `toString()` method in each concrete game object class so that it returns a String describing itself. See the coding notes Appendix (see page 11 below) for additional details, including how to display single decimal value. **Please be conformed to the format specified in above example.**

- Missiles are considered to have special radar that allows them to avoid hitting their own ships, other missiles, and space stations, so we will not worry about handling these collision conditions. Likewise, we will assume that space stations automatically avoid asteroids and NPS.
- The program must handle any situation where the player enters an illegal command – for example, a command to increase the ship speed when no ship has yet been added to the world – by printing an appropriate condition-specific error message on the console (for example, "Cannot execute 'increase' – no ship has been created") and otherwise simply waiting for a valid command.
- The program is not required to have any code that actually checks for collisions between objects; that's something we'll be adding later. For now, the program simply relies on the user to say when such events have occurred, using for example the 'k' and 'c' commands.
- You must follow standard Java coding conventions:
 - class names always start with an upper case letter,
 - variable names always start with a lower case letter,
 - compound parts of compound names are capitalized (e.g., *myExampleVariable*),
 - Java interface names should start with the letter "I" (e.g., *ISteerable*).
- Your program must be contained in a CN1 project called A1Prj. You must create your project following the instructions given at "2 – Introduction to Mobile App Development and CN1" lecture notes posted at Canvas (Steps for Eclipse: 1) File → New → Project → Codename One Project. 2) Give a project name "A1Prj" and check "Java 8 project" 3) Hit "Next". 4) Give a main class name "Starter", package name "com.mycompany.a1", and select a "native" theme, and "Hello World(Bare Bones)" template (for manual GUI building). 5) Hit "Finish".).

Further, you must verify that your program works properly from the command prompt before submitting it to Canvas: First make sure that the A1Prj.jar file is up-to-date. If not, under eclipse, right click on the dist directory and say “Refresh”. Then get into the A1Prj directory and type (all in one line): “java -cp dist\A1Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a1.Starter” for Windows machines (for the command line arguments of Mac OS/Linux machines please refer to the class notes). Alternatively, to prove it is working, you can also run the instructor provided program “RunAssignment.jar” (see class notes). **Substantial penalties will be applied to submissions which do not work properly from the command prompt.**

- You are not required to use any particular data structure to store the game world objects, but *all your game world objects must be stored in a single structure*. In addition, your program must be able to handle changeable numbers of objects at runtime – so you can't use a fixed-size array, and you can't use individual variables.

Consider either the Java *ArrayList* or *Vector* class for implementing this storage. Note that Java Vectors (and ArrayLists) hold elements of type “Object”, but you will need to be able to treat the Objects differently depending on the type of object. You can use the “*instanceof*” operator to determine the type of a given Object, but be sure to use it in a polymorphically-safe way. For example, you can write a loop which runs through all the elements of a world Vector and processes each “movable” object with code like:

```
for (int i=0; i<theWorldVector.size(); i++) {  
    if (theWorldVector.elementAt(i) instanceof IMovable) {  
        IMovable mObj = (IMovable)theWorldVector.elementAt(i);  
        mObj.move();  
    }  
}
```

- Points will be deducted for poorly or incompletely documented programs. Use of JavaDoc-style comments is highly encouraged, but not required.
- Students are encouraged to ask questions or solicit advice from the instructor outside of class. But have your UML diagram ready – it is the first thing the instructor will ask to see.

Deliverables

There are two deliveries for this assignment.

First Delivery: Test cases only. Due Feb 11, 2019

Provide a list of 4 test cases. Each test case must have the following:

1. **Test description:** A description of **a behavior (a scenario)** of the application when a user performs a specific task. This is from a **given user perspective**.
2. **Input actions:** Describe the input actions required to carry out this test (**action performed**).
3. **Expected outputs:** Describe the expected result (**check outcome**) after a test is executed.
4. **Source:** Cite the source information (from this document) of where this test is derived from.

Second Delivery: UML Class Diagram + Source Files + Jar file. Due Feb 25, 2019

There are *three steps* which are required for submitting your program, as follows:

1. Be sure to verify that your program works from the command prompt as explained above.
2. Provide a result of running your test cases. See first delivery's test cases.
3. Create a *single* file in “ZIP” format containing (1) your UML diagram in .PDF format, (2) the entire “src” directory under your CN1 project directory (called A1Prj) which includes source code (“.java”) for all the classes in your program, and (3) the A1Prj.jar located under the “A1Prj/dist” directory which includes the compiled (“.class”) files for your program in a zipped format. Be sure to name your ZIP file as YourLastName-YourFirstName-a1.zip. Also include in your ZIP a text file called “readme” in which you describe any changes or additions you made to the assignment specifications.
3. Login to **Canvas**, select “Assignment 1”, and upload your *verified* ZIP file. Also, be sure to take note of the requirement stated in the course syllabus for keeping a backup copy of all submitted work (save a copy of your ZIP file).

Final notes:

(1) All submitted work must be strictly your own. It will be monitored closely!

(2) A grader will grade only one version of your work. Please check your work closely before submitting to Canvas.

Appendix – CN1 Notes

Input Commands

In CN1, since `System.in` is not supported, we will use a text field located on the form (i.e. the `Game` class) to enter keyboard commands. The `play()` method of `Game` will look like this (we will discuss the details of the GUI and event-handling concepts used in the below code later in the semester):

```
import com.codename1.ui.events.ActionListener;
import com.codename1.ui.Label;
import com.codename1.ui.TextField;
import com.codename1.ui.events.ActionEvent;
import java.lang.String;
private void play()
{
    Label myLabel=new Label("Enter a
Command:");
    this.addComponent(myLabel);
    final TextField myTextField=new TextField();
    this.addComponent(myTextField);
    this.show();
    myTextField.addActionListener(new ActionListener() {

        public void actionPerformed(ActionEvent evt) {

            String sCommand=myTextField.getText().toString();
            myTextField.clear();
            switch (sCommand.charAt(0)) {
                case 'e':
                    gw.eliminate();
                    break;
                    //add code to handle rest of the commands
            } //switch
        } //actionPerformed
    } //new ActionListener()
} //addActionListener
} //play
```

Random Number Generation

The class used to create random numbers in CN1 is `java.util.Random`. This class contains several methods including `nextInt()`, which returns a random integer from the entire range of integers, `nextInt(int)`, which returns a random number between zero (inclusive) and the specified integer parameter (exclusive), and `nextFloat()`, which returns float value (between 0.0 and 1.0). For instance, if you like to generate a random integer value between `x` and `x+y`, you can call `x + nextInt(y)`.

Output Strings

The routine `System.out.println()` can be used to display text. It accepts a parameter of type `String`, which can be concatenated from several strings using the “+”

operator. If you include a variable which is not a String, it will convert it to a String by invoking its `toString()` method. For example, the following statements print out “The value of I is 3”:

```
int i = 3 ;
System.out.println ("The value of I is " + i);
```

Every CN1 class provides a `toString()` method inherited from `Object`. Sometimes the result is descriptive. However, if the `toString()` method inherited from `Object` isn't very descriptive, your own classes should override `toString()` and provide their own String descriptions – including the `toString()` output provided by the parent class if that class was also implemented by you.

For example, suppose there is a class `Book`, and a subclass of `Book` named `ColoredBook` with attribute `myColor` of type `int`. An appropriate `toString()` method in `ColoredBook` might return a description of a colored book as follows:

```
public String toString() {
    String parentDesc = super.toString();
    String myDesc = "color: " + "[" + ColorUtil.red(myColor) + ","
                    + ColorUtil.green(myColor) + ","
                    + ColorUtil.blue(myColor) + "]";
    return parentDesc + myDesc ;
}
```

(Abovementioned static methods of the `ColorUtil` class return the red, green, and blue components of a given integer value that represents a color.)

A program containing a `ColoredBook` called “`myBook`” could then display it as follows:

```
System.out.println ("myBook = " + myBook.toString());
```

or simply:

```
System.out.println ("myBook = " + myBook);
```

Number Formatting

The `System.out.format()` and `System.format()` methods supported in Java are not available in CN1. Hence, in order to display only one digit after the decimal point you can use `Math.round()` function:

```
double dVal = 100/3.0;
double rdVal = Math.round(dVal*10.0)/10.0;

System.out.println("original value: " + dVal);
System.out.println("rounded value: " + rdVal);
```

Above prints the following to the standard output
stream:
original value: 33.333333333333336
rounded value: **33.3** (Please conform to 1 single decimal value).

Assignment #2: Design Patterns and GUIs

Due Dates:

1st Delivery: 4 Test cases – March 14th, 2019 [1 week]

2nd Delivery: UML Class Diagram + Source Files + Test Result + Jar file – April 2nd, 2109 [3 weeks]

Introduction

For this assignment you are to extend your game from Assignment #1 (A1) to incorporate several important *design patterns* and a Graphical User Interface (GUI). The rest of the game will appear to the user to be similar to the one in A1, and most of the code from A1 will be reused, although it will require some modification and reorganization.

An important goal for this assignment will be to reorganize your code so that it follows the *Model-View-Controller (MVC) architecture*. If you followed the structure specified in A1, you should already have a “controller”: the `Game` class containing the `getCommand()` method along with methods to process the various commands (some of which call methods in `GameWorld` when access to the game objects is needed). The `GameWorld` class becomes the “data model”, containing a collection of game objects and other game state values (more details below). You are also required to add two classes acting as “views”: a `points` view which will be graphical, and a `map` view which will retain the text-based form generated by the ‘m’ command in A1 (in A3 we will replace the text-based map with an interactive graphical map).

Most of the keyboard commands from A1 will be replaced by GUI components (menus, buttons, etc.) which generate events. Each such event will have an associated “command” object, and the command objects will perform the same operations as previously performed by the keyboard commands in A1.

The program must use appropriate interfaces for organizing the required design patterns. In all, the following design patterns are to be implemented in this assignment:

- *Observer/Observable* – to coordinate changes in the data with the various views,
- *Iterator* – to walk through all the game objects when necessary,
- *Command* – to encapsulate the various commands the player can invoke,
- *Proxy* – to insure that *views* cannot *modify* the game world.

Model Organization

The “game object” hierarchy will be the same as in A1. `GameWorld` is to be reorganized (if necessary) so that it contains a collection of *game objects* implemented in a single collection data structure. The game object collection is to implement the `Collection` interface and provide for obtaining an `Iterator`. All game objects (moveable and fixed) are to be contained

in this *single* collection data structure¹. The iterator for the collection returns one game object for each call to its `getNext()` method.

The model also contains a few other important pieces of game state data: the current score, number of missiles, elapsed time, and a flag indicating whether Sound is ON or OFF (described later).

Views

A1 contained two functions to output game data: the “m” key for outputting a “map” of the objects in the `GameWorld`, and the “p” key for outputting the “points” (score) information. Each of these operations is to be implemented as a *view* of the `GameWorld` model. To do that, you will need to implement two new classes: a `MapView` class containing code to output the map, and a `PointsView` class containing code to output the points and other state information.

`GameWorld` should therefore be defined as an *observable*, with two *observers* – `MapView` and `PointsView`. Each view should be “registered” as an observer of `GameWorld`. When the controller invokes a method in `GameWorld` that causes a change in the world (such as a game object moving, or a new missile being added) the `GameWorld` notifies its observers that the world has changed. Each observer then automatically produces a new output view of the data it is observing; that is, the game world objects in the case of `MapView` and a description of the point values and related data in the case of `PointsView`. The `MapView` output for this assignment is unchanged from A1: text output on the console. However, the `PointsView` is to present a *graphical* display of the game state values (described in more detail below).

When a change occurs to the model’s data, the model (*observable*) notifies its views (*observers*) and the views then produce a new set of output values. In order for a view to produce the new output, it will need access to some of the data in the model. This access is provided by passing to the observer’s `update()` method a parameter that is a reference back to the model. The view uses that reference to get the data it needs to produce the new output.

Note that providing a view with a reference to the model has the undesirable side-effect that the view has access to the model’s *mutators*, and hence could *modify* model data; see the discussion below on the Proxy design pattern.

Recall that there are two approaches which can be used to implement the **Observer pattern**: defining your own `IObservable` interface, or extending the build-in CN1 `Observable` class. You are required to use the latter approach (where your `GameWorld` class extends `java.util.Observable`). Note that you are also required to use the build-in CN1 `Observer` interface (which also resides in `java.util` package).

GUI Operations

Game class extends `Form` (as in A1) representing the top-level container of the GUI. The `Form` should be divided into three areas: one for commands, one for “points” information, and one for the “map” (which will be an empty `container` for now but in subsequent assignments will be used to display the map in graphical form). See the sample picture at the end. Note that

¹ If you did not implement your game object collection this way in Assignment #1 you must change it for this assignment.

since user input is via GUI components, flow of control will now be event-driven and there is no longer any need to invoke a “`play()`” method – once the `Game` is constructed it simply waits for user-generated input events. Note however that it is still a requirement to have a “`Starter`” class as described in A1.

In A1 your program prompted the player for commands in the form of keyboard input characters. For this assignment the code which prompts for commands and reads keyboard command characters is to be discarded. In its place, commands will be input through three different mechanisms. First, you will need on-screen buttons – one button for each of the input commands from A1 except for the ‘p’ and ‘m’ commands. The program should create the appropriately labeled buttons, add them to the panel, and add the panel as a component of the game. Each button is to have an appropriate command object attached to it, so that when a button gets pushed it invokes the “action performed” method (or “execute” method depending on your approach) in a command object which executes the corresponding code from A1.

The second input mechanism will use CN1 *Key Binding concepts* so that the *left arrow*, *right arrow*, *up arrow*, and *down arrow* keys invoke command objects corresponding to the code previously executed when the “l”, “r”, “t”, and “d” keys (for changing the ship direction and speed) were entered, respectively. The “<” (less than) or “>” (greater than) is used for Player Ship missile launcher to rotate counter clockwise or clockwise. Note that this means that whenever an arrow key is pressed, the program will *immediately* invoke the corresponding action (no “Enter” key press is required). The program is also to use key bindings to bind the SPACE bar to the “fire missile” command and the “j” key to the “jump through hyperspace” command. If you want, you may also use key bindings to map any of the other command keys from A1, but only the ones listed above are required.

The third input mechanism will use a side menu (see class notes for tips on how to create a side menu using CN1 build-in `Toolbar` class). Your GUI should contain one menu: “`File`” containing at least “`New`”, “`Save`”, “`Undo`”, “`Sound`”, “`About`”, and “`Quit`” items; On the “`File`” menu, only the “`Sound`”, “`About`” and “`Quit`” items need to do anything for this assignment (although *all* menu items are required to provide confirmation that they were invoked). The Sound menu item should include a `check box` showing the current state of the “sound” attribute (in addition to the attribute’s state being shown on the `PointsView` GUI panel as described above). Selecting the Sound menu item check box should set a boolean “sound” attribute to “ON” or “OFF”, accordingly. The “`About`” menu item is to display a dialog box (use CN1 build-in `Dialog` class) giving your name, the course name, and any other information you want to display (for example, the version number of the program). “`Quit`” should prompt graphically for confirmation and then exit the program; you should also use a dialog box for this.

Selecting a Command menu item should invoke the corresponding command, just as if the button of the same name had been pushed. Recall that there is a requirement that commands be implemented using the *Command* design pattern. This means that there must be only one of each type of command object, which in turn means that the items on the Command menu must share their command objects with the corresponding control panel buttons. (We could *enforce* this rule using the *Singleton* design pattern, but that is not a requirement in A2; just don’t create more than one of any given type of command object). Each of the commands is to perform exactly the same operations as they did in A1. Please use the following table to confirm your

design work and testing. The 'X' symbol means selected. A blank selection cell means it is not applicable.

A2 Command Options (required for A2)

Command Label (Display in GUI)	Command	Control Panel	Side Menu	Key Binding
+ Asteroid	Add Asteroid	X		
+ NPS	Add Non Player Ship	X		
+ Space Station	Add Space Station	X		
+ PS (1)	Add Player Ship	X		
PS Speed (+)	Increase Player Ship Speed	X		Up Arrow, i
PS Speed (-)	Decrease Player Ship Speed	X		Down Arrow, d
PS Left	Turn left Player Ship	X		l, Left Arrow
PS Right	Turn right Player Ship	X		r, Right Arrow
MSL Left	Turn Left Missile Launcher	X		<
MSL Right	Turn right Missile Launcher	X		>
PS Fire	Player Ship firing	X		Space Bar
NPS Fire	Non Player Ship firing	X		
Jump	Jump	X		j
Load PS	Load Missiles to Player Ship	X		
PS Missile (Asteroid)	Player Ship Missile hits Asteroid	X		
PS Missile (NPS)	Player Ship Missile hits Non Player Ship	X		
NPS Missile (PS)	Non Player Ship Missile hits Player Ship	X		
PS (Asteroid)	Player Ship hits Asteroid	X		
PS (NPS)	Player Ship hits Non Player Ship	X		
Asteroid (Asteroid)	Asteroid hits another Asteroid	X		
Asteroid (NPS)	Asteroid hits Non Player Ship	X		
Tick	Run a Tick command	X		

Quit	Quit the game		X	q
About	About		X	
Sound	Sound Check Box		X	
New	New		X	
Save	Save		X	
Undo	Undo		X	
Map	print a “map”			m
Print	print a display			p

The `PointsView` class should display game state data for each of the points elements from A1 (total points, number of missiles, and elapsed time), plus one new attribute: “`Sound`” with value either ON or OFF.² As described above, `PointsView` must be registered as an observer of `GameWorld`. Whenever any change occurs in `GameWorld`, the `update()` method in its observers is called. In the case of the `PointsView`, what `update()` does is update the contents of the labels displaying the game state (use `Label` method `setText(String)` to update the label). Note that these are exactly the same point values as were displayed previously (with the addition of the “`Sound`” attribute); the only difference now is that they are displayed *graphically* in `PointsView`.

Although we cover most of the GUI building details in the lecture notes, there will most likely be some details that you will need to look up using the CN1 documentation (i.e., CN1 JavaDocs and developer guide). It will become increasingly important that you familiarize yourself with and utilize these resources.

Observer/Observable Pattern

Please refer to the `View` section above.

Command Design Pattern

The approach you must use for implementing command classes is to have each command extend the CN1 build-in Command class (which implements the `ActionListener` interface), as shown in the course notes. Code to perform the command operation then goes in the command’s `actionPerformed()` method. Hence, `actionPerformed()` method of each command class that performs an operation invoked by a single-character command in A1, should call the appropriate method in the `GameWorld` that you have implemented in A1 when related single-character command is entered from the text field (e.g., accelerate command’s `actionPerformed()` would call `accelerate()` method in `GameWorld`).

² In this version of the game there will not actually be any sound; just the state value ON or OFF (a boolean attribute that is *true* or *false*). We’ll see how to actually add sound later.

Hence, most command objects need to have the GameWorld as its target since they need to access the methods defined in this class. You could implement the specification of a command's target either by passing the target (reference to GameWorld) to the command constructor, or by including a "setTarget()" method in the command.

Button class is automatically able to be a "holder" for command objects; Button have a setCommand() method which allows inserting a command object into the button. Command automatically becomes a listener when added to a Button via setCommand() (you do not need to also call addActionListener()), and the specified Command is automatically invoked when the button is pressed, so if you use the CN1 facilities correctly then this particular observer/observable relationship is taken care of automatically.

The Game constructor should create a single instance of each command object (for example, a "Accelerate" command object, etc.), then insert the command objects into the command holders (buttons, side menu items, title bar area items) using methods like setCommand() (for buttons), addCommandToSideMenu() (for side menu items), and addCommandToRightBar() (for title bar area items). You should also bind these command objects to the keys using addKeyListener() method of Form. You must call super("command_name") in the constructors of command objects by providing appropriate command names. These command names will be used to override the labels of buttons and/or to provide the labels of side menu items or title bar area item(s).

Note that commands can be invoked using multiple mechanisms (e.g., from a keystroke and from a button); it is a requirement that only one command object be implemented for each command and that the same command object be invoked from each different command holder. As a result, it is important to make sure that nothing in any command object contains knowledge of how (e.g., through which mechanism) the command was invoked.

Each class which extended the CN1 build-in Command class must be resided in a separated single file.

Iterator Design Pattern

The game object collection must be accessed through an appropriate implementation of the Iterator design pattern. That is, any routine that needs to process the objects in the collection must not access the collection directly, but must instead acquire an iterator on the collection and use that iterator to access the objects. You should develop your own interfaces to implement this design pattern, instead of using the related build-in CN1 interfaces. The game object collection will implement an interface called **ICollection** which defines at least two methods: for adding an object to the collection (i.e., **add()**) and for obtaining an iterator over the collection (i.e., **getIterator()**). The iterator should exist as a private inner class inside game object collection class and should implement an interface called **Iterator** which defines at least two methods: for checking whether there are more elements to be processed in the collection (i.e., **hasNext()**) and returning the next element to be processed from the collection (i.e., **getNext()**).

Note however the following implementation requirement: the game object collection must provide an iterator completely implemented by you. Even if the data structure you use has an iterator provided by CN1, you must implement an iterator yourself: The iterator should keep track of the current index and it cannot make use of the built-in iterator defined for the data structure

(e.g., you cannot call **hasNext()** method of the built-in iterator defined for **Vector**/**ArrayList** from the **hasNext()** method of your iterator class). However, the iterator can use following build-in methods of the data structure: **size()** and **elementAt()/get()**.

Proxy Design Pattern

To prevent views from being able to modify the **GameWorld** object received by their **update()** methods, the model (**GameWorld**) should pass a GameWorld proxy to the views; this proxy should allow each view to obtain all the required data from the model while prohibiting any attempt by the view to *change* the model. The simplest way to do this is to define an interface **IGameWorld** listing the methods provided by a **GameWorld**, and to provide a new class **GameWorldProxy** which implements this same interface. The **GameWorld** model then passes to each observer's **update()** method a **GameWorldProxy** object instead of the actual **GameWorld** object. See the attachment at the end for additional details.

Recall that there are two approaches which can be used to implement the Observer pattern: defining your own **IObservable** interface, or extending the build-in CN1 Observable class. You are required to use the latter approach (where your GameWorld class extends `java.util.Observable`). Note that you are also required to use the build-in CN1 Observer interface (which also resides in `java.util` package).

Additional Notes

- All menu items, buttons, and other GUI components must display a message on the console indicating they were selected. You may add additional GUI items if you like, but if you do, they too must at least indicate on the console that they were selected.
- You must use BorderLayout as the layout manager of your form and use FlowLayout or BoxLayout for the containers you have added to different areas of your form. These layout managers automatically place and size the containers/components that you have added to your form/containers.
- You can change the size of your buttons/labels using **setPadding()** method of Style. Each label in PointView should be divided into two parts, text and value, and padding should be added to the value part so that the labels look stable when value changes as discussed in the class notes. You can also use **setPadding()** on left and right control containers to start adding buttons at positions which are certain pixels below their upper borders.
- In A2, you must assign the size of your game world by querying the size of your MapView container (instead of assigning your width and height to 1024x768 as in A1, assign them to width and height of MapView) using **getWidth()** and **getHeight()** method of Component as discussed in the class notes.
- You must change the style of your buttons using methods like **setBgTransparency()** , **setBgColor()** , **setFgColor()** of **Style** class so that they look different from labels as discussed in the class notes. You can extend from the built-in **Button** class and do the styling in this new class. Then, while you construct your GUI, instead of creating instances of the build-in **Button** class, you can create objects out of this new user-defined button class.

- Note that all game data manipulation by a command is accomplished by the command object invoking appropriate methods in the **GameWorld** (the *model*). Note also that every change to the **GameWorld** will invoke *both* the **MapView** and **PointsView** observers – and hence generate the output formerly associated with the “m” and “d” commands. This means you do not need the “d” or “m” commands; the output formerly produced by those commands is now generated by the observer/observable process.
- Programs must contain appropriate documentation as described in A1 and in class.
- Note that since the ‘tick’ command causes moveable objects to move, every ‘tick’ will result in a new map view being output (because each tick changes the model). Note however that it is *not* the responsibility of the ‘tick’ command code to produce this output; it is a side effect of the observable/observer pattern. Note also that this is not the only command which causes generation of output as a side effect. You should verify that your program correctly produces updated views automatically whenever it should.
- You may not use a “GUI Builder” tool for this assignment. (If you don’t know what a GUI Builder is, don’t worry about it.)
- All functionality from the previous assignment must be retained unless it is explicitly changed or deleted by the requirements for this assignment. The one function which renewed is the direction of the Steerable Missile Launcher. In A2, a user can control the direction of the Steerable Missile Launcher by revolving it about the center of the ship in a counterclockwise direction or clockwise. This direction can be different from the ship navigation direction. User can select this function by the using the “MSL Left” (or “MSL Right”) command - to appear on the command menu. The key binding value for this command is “<” (or “>”). This command turns the Steerable Missile Launcher by small θ angle. You decide a value.
- As before, you should develop a *UML diagram* showing the relationships between your classes, including not only the major fields and methods required but also the interfaces and the relationships between classes using those interfaces (for example, Observer/Observable). This will be particularly useful in helping you understand what modifications you need to make to your code from A1. Feel free to see me if you are not sure your design is correct. Note that if you come to see me for help with the program – which is encouraged – the first thing I am likely to ask is to see is your UML diagram.
- Although the **MapView** Container is empty for this assignment, you should put a border around it and install it in the frame, to at least make sure that it is there. See the sample picture below.

Deliverables

There are two deliveries for this assignment.

First Delivery: Test cases only. Due March 14th, 2019

Provide a list of 4 test cases. Each test case must have the following:

1. **Test description:** A description of **a behavior (a scenario)** of the application when a user performs a specific task. This is from a **given user perspective**.
2. **Input actions:** Describe the input actions required to carry out this test (**action performed**).
3. **Expected outputs:** Describe the expected result (**check outcome**) after a test is executed. These outputs **MUST BE** verifiable.
4. **Source:** Cite the source information (from this document) of where this test is derived from.

Second Delivery: UML Class Diagram + Test Result + Source Files + Jar file. Due April 2nd , 2019

There are *four steps* which are required for submitting your program, as follows:

1. Be sure to verify that your program works from the command prompt as explained above.
2. Provide a result of running your test cases. See first delivery's test cases.
3. Create a *single* file in “ZIP” format containing (1) your UML diagram in .PDF format, (2) the entire “src” directory under your CN1 project directory (called A2Prj) which includes source code (“.java”) for all the classes in your program, and (3) the A2Prj.jar located under the “A2Prj/dist” directory which includes the compiled (“.class”) files for your program in a zipped format. Be sure to name your ZIP file as YourLastName-YourFirstName-a2.zip. Also include in your ZIP a text file called “readme” in which you describe any changes or additions you made to the assignment specifications.
4. Login to **Canvas**, select “Assignment 2”, and upload your *verified* ZIP file. Also, be sure to take note of the requirement stated in the course syllabus for keeping a backup copy of all submitted work (save a copy of your ZIP file).

Final notes:

(1) All submitted work must be strictly your own. It will be monitored closely!

(2) A grader will grade only one version of your work. Please check your work closely before submitting to Canvas.

Sample GUI

The following shows an example of what your game GUI **MIGHT** look like. Notice that it has a control panel on the left containing all the required buttons, a menu bar containing the required menus, a PointsView panel near the top showing the current game state information,

and an (empty) MapView panel in the middle for future use. The title “Asteroids Game” displays at the top.



(Note: This is a mockup screen only. Please do not use this GUI for your Spring 2019 Assignments)

Java Notes

Below is the organization for your MVC code for A2. Note that in this organization, we are using the CN1 “Observable” class and CN1 “Observer” interface to derive your GameWorld and views. Doing it this way has the benefit of CN1 handling the “list of observers” for you. Note also that this pseudo-code shows one way of registering Observers with Observables: having the controller handle the registration. It is also possible to have each Observer handle its own registration in its constructor (examples are shown in the course notes). You may use either approach in your program.

```
public class Game extends Form {  
  
    private GameWorld gw;  
    private MapView mv; // new in A2  
    private PointsView pv; // new in A2  
  
    public Game() {  
        gw = new GameWorld(); // create "Observable"  
        mv = new MapView(); // create an "Observer" for the map
```

```

    pv = new PointsView(gw);      // create an "Observer" for the points
    gw.addObserver(mv);          // register the map Observer
    gw.addObserver(pv);          // register the points observer

    // code here to create menus, create Command objects for each command,
    // add commands to Command menu, create a control panel for the buttons,
    // add buttons to the control panel, add commands to the buttons, and
    // add control panel, MapView panel, and PointsView panel to the form

    this.show();
}
}

public interface IGameWorld {
    //specifications here for all GameWorld methods
}

public class GameWorld extends Observable implements IGameWorld {
    // code here to hold and manipulate world objects, handle observer registration,
    // invoke observer callbacks by passing a GameWorld proxy, etc.
}

public class GameWorldProxy extends Observable implements IGameWorld {
    // code here to accept and hold a GameWorld, provide implementations
    // of all the public methods in a GameWorld, forward allowed
    // calls to the actual GameWorld, and reject calls to methods
    // which the outside should not be able to access in the GameWorld.
}

public class MapView extends Container implements Observer {
    public void update (Observable o, Object arg) {
        // code here to output current map information (based on the data in the Observable)
        // to the console. Note that the received "Observable" is a GameWorld PROXY and can
        // be cast to type IGameWorld in order to access the GameWorld methods in it.
    }
}

public class PointsView extends Container implements Observer {
    public void update (Observable o, Object arg) {
        // code here to update labels from data in the Observable (a GameWorldPROXY)
    }
}

```

Assignment #3: Interactive Graphics and Animation

Due Date: Tuesday, April 30th [3 Weeks]

Introduction

The purpose of this assignment is to help you gain experience with interactive graphics and animation techniques such as repainting, timer-driven animation, collision detection, and object selection. Specifically, you are to make the following modifications to your game:

- (1) the game world map is to display in the GUI, rather than in text form on the console,
- (2) movement (animation) of game objects is to be driven by a timer,
- (3) the game is to support dynamic collision detection and response,
- (4) the game is to include sounds appropriate to collisions and other events, and
- (5) the game is to support simple interactive editing of some of the objects in the world.

1. Game World Map

If you did assignment #2 properly, your program included an instance of a **MapView** class that is an observer that displayed the game elements on the console. **MapView** also extended **Container** and that container was placed in the middle of the game frame, although it was empty.

For this assignment, **MapView** will display the contents of the game *graphically* in the **Container** in the middle of the game screen. When the **MapView update()** is invoked, it should now call **repaint()** on itself. As described in the course notes, **MapView** should also implement (override) **paint()**, which will therefore be invoked as a result of calling **repaint()**. It is then the duty of **paint()** to iterate through the **GameWorld** objects (via an iterator, as before) invoking **draw(Graphics g)** in each **GameWorld** object – thus redrawing all the objects in the world in the container. Note that **paint()** must have access to the **GameWorld**. This means that a reference to the **GameWorld** must be saved when **MapView** is constructed, or else the **update()** method must save it prior to calling **repaint()**. Note that the modified **MapView** class communicates with the rest of the program *exactly* as it did previously (e.g. it is an observer of its observable; it gets invoked via a call to its **update()** method as before; etc.).

- Map View Graphics

As specified in assignment #1, each game object has its own different graphical representation (shape). The appropriate place to put the responsibility for drawing each shape is within each type of game object (that is, to use a *polymorphic* drawing capability). The program should define a new interface named (for example) **IDrawable** specifying a method **draw(Graphics g, Point pCmpRelPrnt)**. Each game object should then implement the **IDrawable** interface with code that knows how to draw that particular object using the received “**Graphics**” object (which belong to **MapView**) and component location (**MapView**’s origin location which is located at its the upper left corner) relative to its parent container’s origin (parent

of MapView is the content pane of the Game form and origin of the parent is also located at its upper left corner). Remember that calling `getX()` and `getY()` methods on MapView would return the MapView component's location relative to its parent container's origin.

Each object's `draw()` method draws the object in its current color and size, at its current location. Recall that current location of the object is defined relative to the origin of the game world (which corresponds to the origin of the MapView in A#2 and A#3). Hence, do not forget to add MapView's origin location (relative to its parent container's origin) to the current location while drawing your game objects since `drawXXX()` methods of Graphics expects coordinates which are relative to the parent container's origin.

Each concrete game object is to have a unique shape – for example, a player ship could be represented as an unfilled (open) triangle; a non player ship could be represented filled (close) triangle; a space station could be represented as a filled (solid) circle when it is “on” and an unfilled circle when it is “off”; a missile could be represented as a narrow filled rectangle while an asteroid could be represented as an unfilled square. (The preceding examples are not explicit requirements; you may choose other representations as long as each kind of object is unique on the screen.) Note in particular the requirement for a space station to have different representations depending on its “blink state”.

Recall that the location of each object is the location of the center of that object. Each `draw()` method must take this definition into account when drawing an object. Remember that the `drawXXX()` method of the Graphics class expects to be given the X,Y coordinates of the upper left corner of the rectangle to be drawn, so for instance, a `draw()` method for a rectangular object would need to use the location, and size attributes of the object to determine where to draw the rectangle so its center coincides with its location. For example, the X coordinate of the upper left corner of a rectangle is $(\text{center.x} - \text{width}/2)$ relative to the origin of the game world. Similar adjustments apply to drawing circles.

2. Animation Control

The Game class is to include a timer (you should use the UITimer, a build-in CN1 class) to drive the animation (movement of movable objects). Game should also implement Runnable (a build-in CN1 interface). Each tick generated by the timer should call the `run()` method in Game. `run()` in turn can then invoke the “Tick” command from the previous assignment, causing all moveable objects to move. This replaces the “Tick” button, which is no longer needed and should be eliminated.

There are some changes in the way the Tick command works for this assignment. In order for the animation to look smooth, the timer itself will have to tick at a fairly fast rate (**about every 20 msec or so**). In order for each movable object to know how far it should move, each timer tick should pass an “elapsed time” value to the `move()` method. The `move()` method should use this elapsed time value when it computes a new location. For simplicity, you can simply pass the value of the timer event rate (e.g., 20 msec), rather than computing a true elapsed time. However, it is a requirement that each `move()` computes movement based on the value of the `elapsedTime` parameter passed in, not by assuming a hard-coded time value within the `move()` method itself. You should experiment to determine appropriate movement values (e.g., in A#1, we have specified the initial speed of an object to be a random value between 0 and 15, you may need to adjust this range to make your objects have reasonable which is not to too fast or too slow). In addition, be aware that methods of the built-in CN1 Math class that you will use in `move()` method (e.g., `Math.cos()`, `Math.sin()`) expects the angles to be provided in radians not

degrees. You can use `Math.toRadians()` to convert degrees to radians. Likewise, the built-in `MathUtil.atan()` method that you might have used in the strategy classes also produces an angle in radians. You can use `Math.toDegrees()` to convert degrees to radians. Remember that a `UITimer` starts as soon as its `schedule()` method is called. To stop a `UITimer` call its `cancel()` method. To re-start it call the `schedule()` method again.

3. Collision Detection and Response

There is another important thing that needs to happen each time the timer ticks. In addition to updating the Elapsed Time as necessary and invoking `move()` for all movable objects, your code must tell the game world to determine if there are any collisions between objects, and if so to perform the appropriate “collision response”. The appropriate way to handle collision detection/response is to have each kind of object which can be involved in collisions implement a new interface like “**ICollider**” as discussed in class and described in the course notes. That way, colliding objects can be treated polymorphically.

In the previous assignment, collisions were caused by pressing one of the buttons (“kill asteroid”, “crash ship”, etc.), and the objects involved in the collision were chosen arbitrarily. Now, the type of collision will be detected automatically during collision detection, so the “load missiles”, “kill asteroid”, “crash ship”, and “exterminate” buttons are no longer needed; they should be removed and replaced with collision detection which checks for the corresponding collisions. Collision detection will require objects to check to see if they have collided with other objects, so the actual collisions will no longer be arbitrary, but will correspond to actual collisions in the game world.

4. Commands

Please use the following table to confirm your design work and testing. The ‘X’ symbol means selected. A blank selection cell means it is not applicable.

A3 Command Options (required for A3)

Command Label (Display in GUI)	Command	Control Panel	Side Menu	Key Binding
+ Asteroid	Add Asteroid	X		
+ Space Station	Add Space Station	X		
+ PS (1)	Add Player Ship	X		
PS Speed (+)	Increase Player Ship Speed	X		Up Arrow, i
PS Speed (-)	Decrease Player Ship Speed	X		Down Arrow, d
PS Left	Turn left Player Ship	X		l, Left Arrow
PS Right	Turn right Player Ship	X		r, Right Arrow
MSL Left	Turn Left Missile Launcher	X		<
MSL Right	Turn right Missile Launcher	X		>

PS Fire	Player Ship firing	X		Space Bar
Jump	Jump	X		j
Load PS	Load Missiles to Player Ship	X		
Quit	Quit the game		X	Q
About	About		X	
Sound	Sound Check Box		X	
New	New		X	
Save	Save		X	
Undo	Undo		X	
Play/Pause	Play/Pause game	X		
Refuel	Refuel Missile	X		

Collision response (that is, the specific action taken by an object when it collides with another object) will be similar as before: when two asteroids collided, they are removed¹; when a player ship collides with an asteroid, the ship is removed (along with the asteroid), and a life is lost, and if there are lives remaining a new ship should be created². Otherwise, the game should prompt a player to see if he/she would like to play the game again; when a non-player ship collides with an asteroid, the ship and the asteroid are removed; when a missile collides with an asteroid the missile and asteroid are removed and the player scores some points; when the ship collides with a space station the ship receives a new missile supply. Some collisions also generate a *sound* (see below). There are more hints regarding collision detection in the notes below.

5. Sound

You may add as many sounds into your game as you wish. However, you must implement particular, clearly different sounds for *at least* the following situations:

- (1) when a ship fires a missile,
- (2) when a PS missile launcher rotates (i.e. sound of a robot moving part, very briefly),
- (3) when a missile collides with an asteroid,
- (4) when the game ends due to no more lives, and
- (5) some sort of appropriate background sound that loops continuously during animation.

You may also add sounds for other events if you like. Sounds should only be played if the “SOUND” attribute is “ON”. You may use any sounds you like, as long as I can show the game to the Dean/Chair (in other words, the sounds are not disgusting or obscene). Short, unique sounds tend to improve game playability by avoiding too much confusing sound overlap. Do not

¹ An acceptable variation for asteroid/asteroid collision handling would be to remove the colliders and then *add to the world four new asteroids, each smaller in size and located near the originals*. This more closely simulates the original game. This is not a requirement, however.

² If your previous assignment did not add a new ship when there are remaining lives, you must modify your program for this assignment to do so.

use copyrighted sounds. You may search the web to find these non-copyrighted sounds (e.g., www.findsounds.com).

You must copy the sound files directly under the src directory of your project for CN1 to locate them. You should add Sound and BGSound classes to your project to add a capability for playing regular and looping (e.g., background) sounds, respectively, as discussed in the lecture notes. These classes encapsulate given sound files by making use of InputStream, MediaManager, and Media build-in CN1 classes. In addition to these built-in classes, BGSound also utilizes Runnable build-in CN1 interface. You should create a single sound object for each audio file and when you need to play the same sound file you should use this single instance.

6. Object Selection and Game Modes

In order for us to explore the Command design pattern more thoroughly, and to gain experience with graphical object selection, we are going to add an additional capability to the game. Specifically, the game is to have two modes: “play” and “pause”. The normal game play with animation as implemented above is “play” mode. In “pause” mode, animation stops – the game objects don’t move and the background looped sound also stops. Also, when in pause mode, the user can use the mouse to select some of the game objects.

Ability to select the game mode should be implemented via a new GUI command button that switches between “play” and “pause” modes. When the game first starts it should be in the play mode, with the mode control button displaying the label “Pause” (indicating that pushing the button switches to pause mode). Pressing the Pause button switches the game to pause mode and changes the label on the button to “Play”, indicating that pressing the button again resumes play and puts the game back into play mode (also restarting the background sound).

- Object Selection

When in pause mode, the game must support the ability to interactively *select* objects. The appropriate mechanism for identifying “*selectable*” objects is to have those objects implement an interface such as **ISelectable** which specifies the methods required to implement selection, as discussed in class and described in the course notes. Selecting an object allows the user to perform certain actions on the selected object. Each selected object must be highlighted in some way (you may choose the form of highlighting, as long as there is some visible change to the appearance of each selected object). For this assignment, only *asteroids*, *Non Player Ships*, and *missiles* are selectable. Selection is impossible in play mode.

An individual object is selected by pressing the pointer on it. Pressing on an object selects that object and “unselects” all other objects. Clicking in a location where there are no objects causes the selected object to become unselected. Remember that pointer (x,y) location received by overriding the *pointerPressed()* method of *MapView* is relative to screen origin. You can make this location relative to *MapView*’s parent’s origin by calling the following lines inside the *pointerPressed()* method:

```
x = x - getParent().getAbsoluteX()  
y = y - getParent().getAbsoluteY()
```

A new **Refuel** command (Action) is to be added to the game, invocable from a new “Refuel” GUI button. When the Refuel command is invoked, a *selected* missile is to have its fuel level set to the maximum. The Refuel action should only be available while in pause mode, and should have no effect on unselected items.

- Command Enabling/Disabling

Commands should be enabled only when their functionality is appropriate. For example, the Position command should be disabled while in play mode; likewise, commands that involve playing the game (e.g., changing the player's car direction) should be disabled while in pause mode. Note that disabling a command should disable it for all invokers (buttons, keys, and menu items). Note also that a disabled button or menu item should still be visible; it just cannot be active (enabled). This is indicated by changing the appearance of the button or menu item. To disable a button and menu item (which is added to the form by addCommandToSideMenu()), use setEnabled() methods of Button and Command classes, respectively. To disable a key, use removeKeyListener() method of Form (remember to re-add the key listener when the command is enabled). You can set disabled style of a button using getDisabledStyle().setXXX() methods on the Button.

7. Game Randomization

To keep the game excitement, we would like to have a Non-Player Ship (NPS) to randomly enter the game world and shoot at the Player Ship (PS). As per assignment 1, the NPS size, speed, direction, and location are randomized. To control the creation of the NPS, we can seed this in the run method (see section 2 above). A code in the run method can have the following logic:

```
int roll = ClassName.genRandInt(min, max);
```

and

```
if (roll >= smallValue && roll <= bigValue) {  
    gw.addNPS(); // add a non-player ship to the world  
}
```

The genRandInt method can be coded as follows:

```
public static int genRandInt(int min, int max) {  
    Random r = new Random();  
    int x = r.nextInt((max - min) + 1) + min;  
    return x;  
}
```

Additional Notes

- Please make sure that in A#2 you have added a MapView object directly to the center of the form. Adding a Container object to the center and then adding a MapView object onto it would cause incorrect results when a default layout is used for this center container (e.g., you cannot see any of the game objects being drawn in the center of the form).
- To draw a un-filled and filled triangle you can use drawPolygon()/fillPolygon() methods of Graphics. Note that drawArc(x, y, 2*r, 2*r, 0, 360) draws a circle with radius r at location (x,y).

- As before, the origin of the game world (which corresponds to the origin of the MapView for now) is considered to be in its lower left corner. Hence, the Y coordinate of the game world grows upward (Y values increase upward). However, origin of the MapView container is at its upper left corner and Y coordinate of the MapView grows downward. So when a game object moves north (e.g., its heading is 0 and hence, its Y values is increasing) in the game world, they would move up in the game world. However, due to the coordinate systems mismatch mentioned above, heading up in the game world will result in moving down on the MapView (screen). Hence your game will be displayed as upside down on the screen. We will fix it in A#4.
- The simple shape representations for game objects will produce some slightly weird visual effects in the map view. For example, squares or triangles will always be aligned with the X/Y axes even if the object being represented is moving at an angle relative to the axes. This is acceptable for now; we will see how to fix it in A#4.
- You may adjust the size of game objects for playability if you wish; just don't make them so large (or small) that the game becomes unwieldy.
- As indicated in A#2, boundaries of the game world are equal to dimensions of MapView. Hence, moveable object should consider the width and the height of the MapView in their move() method not to be out of boundaries.
- Because the sound can be turned on and off by the user (using the menu), and also turns on/off automatically with the pause/play button, you will need to test the various combinations. For example, pressing pause, then turning off sound, then pressing play, should result in the sound not coming back on. There are several sequences to test.
- When two objects collide handling the collision should be done only once. For instance, when two asteroids collided (i.e., a1 and a2) the asteroids damage level should be increased only once (example only), not twice. In the two nested loops of collision detection, a1.collidesWith(a2) and a2.collidesWith(a1) will both return true. However, if we handle the collision with a1.handleCollision(a2) we should not handle it again by calling a2.handleCollision(a1). This problem is complicated by the fact that in most cases the same collision will be detected repeatedly as one object passes through the other object. Another complication is that more than two objects can collide simultaneously. One straight-forward way of solving this complicated problem, is to have each collidable object (that may involve in such a problem) keep a list of the objects that it is already colliding with. An object can then skip the collision handling for objects it is already colliding with. Of course, you'll also have to remove objects from the list when they are no longer colliding. Implementation of this solution would require you to have a Vector (or ArrayList) for each collidable object (i.e., car object) which we will call as "collision vector". When collidable object obj1 collides with obj2, right after handling the collision, you need to add obj2 to collision vector of obj1. If obj2 is also a collidable object, you also need to add obj1 to collision vector of obj2. Each time you check for possible collisions (in each clock tick, page 7 of 8 after the moving the objects) you need to update the collision vectors. If the obj1 and obj2 are no longer colliding, you need to remove them from each other's collision vectors. You can use remove() method of Vector to remove an object from a collision vector. If two objects are still colliding (passes through each other), you should not add them again to the collision vectors. You can use contains() method of

Vector to check if the object is already in the collision vector or not. The contains() method is also useful for deciding whether to handle collision or not. For instance, if the collision vector of obj2 already contains obj1 (or collision vector of obj1 already contains obj2), it means that collision between obj1 and obj2 is already handled and should not be handled again.

- You should tweak the parameters of your program for playability after you get the animation working. Things that impact playability include the screen size, object sizes and speeds, moving speed of objects, collision distance, etc. Your game is expected to operate in a reasonably-playable manner.
- As before, you may not use a “GUI Builder” for this assignment.
- All functionality from the previous assignment must be retained unless it is explicitly changed or deleted by the requirements for this assignment.
- When the game is over, animation should stop, and a dialog box should display showing the player’s final score.

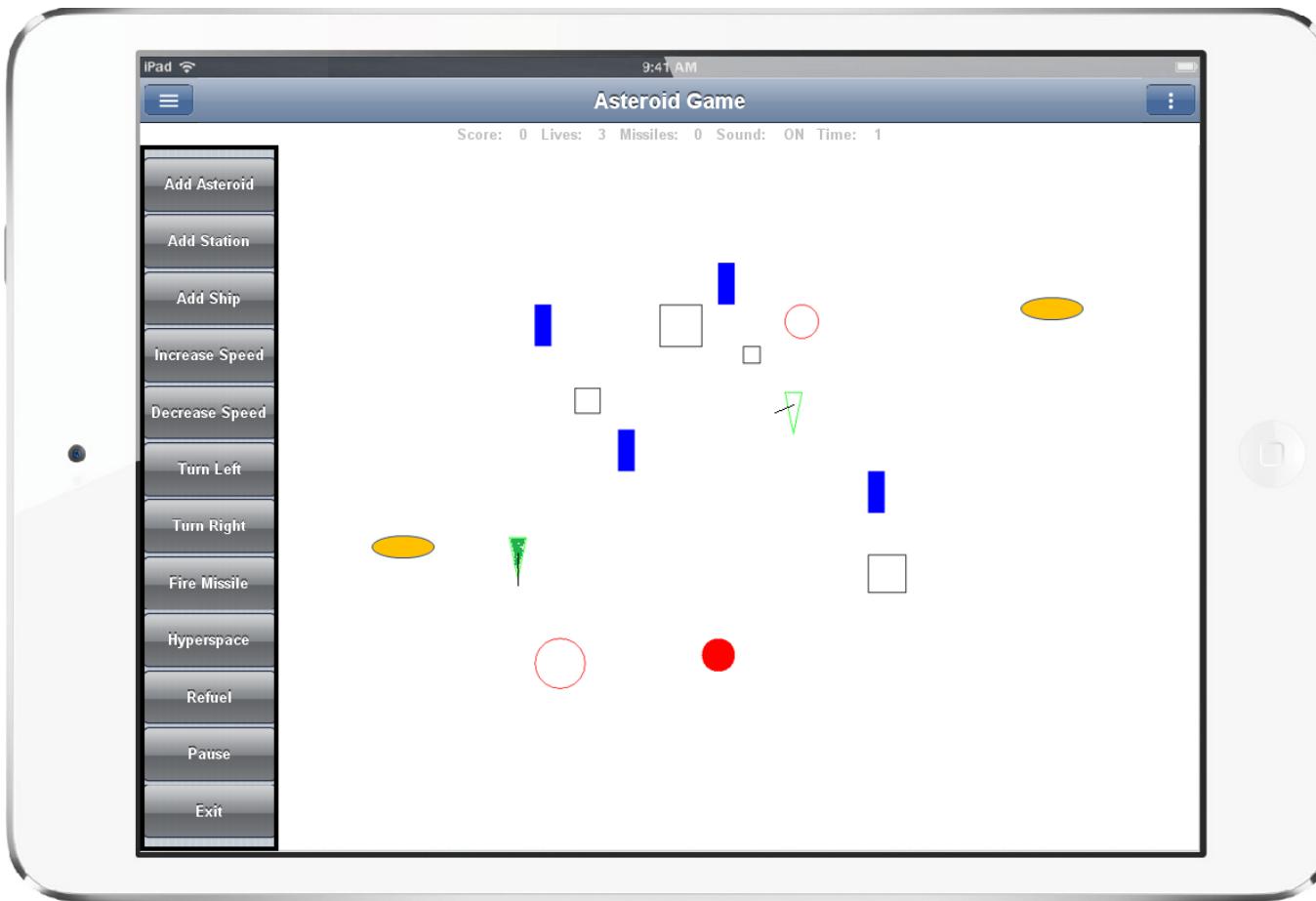
8. Deliverables

Submitting your program requires the same three steps as for A#1 and A#2 except that you do not need to submit a UML for A#3: 1. Be sure to verify that your program works from the command prompt as explained above. 2. Create a single file in “ZIP” format containing (1) the entire “src” directory under your CN1 project directory (called A3Prj) which includes source code (“.java”) for all the classes in your program and the audio files, and (2) the A3Prj.jar located under the “A3Prj/dist” directory which includes the compiled (“.class”) files for your program and audio files in a zipped format. Be sure to name your ZIP file as YourLastName-YourFirstName-a3.zip. Also include in your ZIP a text file called “readme” in which you describe any changes or additions you made to the assignment specifications. 3. Login to Canvas, select “Assignment 3”, and upload your verified ZIP file. Also, be sure to take note of the requirement stated in the course syllabus for keeping a backup copy of all submitted work (save a copy of your ZIP file).

All submitted work must be strictly your own! (There is NO REDO for A3)

9. Sample GUI

The following shows one example of how a completed A3 game **MIGHT** look. Notice that it has a control panel on the left with the required command buttons (including the disabled Refuel command since the game here is in “Play” mode as indicated by the fact that the Pause/Play button shows “Pause”). It also has “File” and “Commands” menus, a points view panel showing the current game state, and a map view panel containing 4 asteroids (black squares), 4 missiles (blue rectangles), 3 space stations (red circles; two “off” and one “on”) and one PS (hollow green triangle), with a rotatable missile launcher (pointing away from the ship heading). A NPS is a solid triangle with a Missile Launcher points in the same heading direction. Note that the moving objects are not necessarily oriented (rotated) in the direction in which they are moving; this is acceptable for this assignment (we’ll see how to take care of that later). Note also that the world is “upside down” – the ship is currently heading “north” (0 degrees) but is facing the *bottom*. We will fix this in a subsequent assignment as well.



(Note: This is a mockup screen only. Please do not use this GUI for your A3, Spring 2019 Assignment)

Assignment #4: Transformations and Bezier Curves

(Bonus assignment)

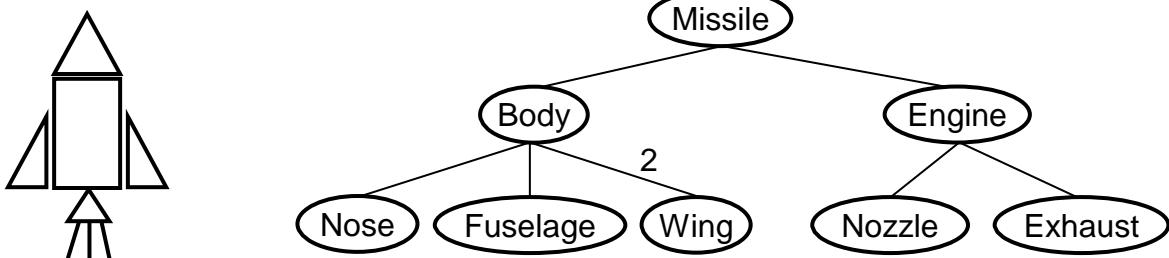
Due Date: Friday, May 10th

Overview

The objective of this assignment is to extend your program from Assignment #3 (A3) to include 2D transformations and Bezier curves. As always, all operations from previous assignments must continue to work as before (except where superseded by new operations). In this assignment, you are to add the following three things:

1. Local, world, and device coordinate systems. Objects are drawn in their “local” coordinates; transformations are then used to map drawn objects to “world” and then to “screen” coordinates. The game world is defined by an independent *world coordinate system*. Initially, the origin (0,0) position of the world coincides with the lower left corner of the screen (MapView area). A screen transformation is used so that the game world appears “right-side up”. The initial (default) “world window” should match the screen size in your previous assignments, so that initially the entire game world is visible on the screen.

2. Hierarchical and dynamic object transformations. Missiles are to be drawn as dynamically transformable hierarchical objects.¹ Each missile is to be composed of a hierarchy of at least two sub-levels of shapes, each with its own transformation which positions the shape in relation to its parent. Additionally, one of the shape transformations must change dynamically as a function of the timer. For example, a missile could be constructed as shown in the following figure, made up of a body containing a fuselage, nose-cone, and wings, and an engine made up of a nozzle and exhaust, with the wings moving in and out or the exhaust vibrating as it flies. (The figure is just an example; you may construct your own hierarchical design as long as it meets the above requirements.) Transformations should also include the appropriate rotation so that ships and missiles face the direction they move. Additionally, asteroids must also “tumble” (rotate around their top-level origin) as they move through space.



¹ You may also make other objects hierarchical, but missiles must be; this is in order to test that *selection* works with a *hierarchical* object.

3. Bezier Curves. The player is to be given the ability to throw *plasma* waves from ships. Plasma waves are movable objects which happen to be shaped like Bezier curves. When the user presses the 'P' key it causes the ship to throw a plasma wave. Plasma waves move in the direction the ship is heading at the moment the wave is generated. When a plasma wave collides with an asteroid, the asteroid is immediately destroyed (removed from the game) and the user scores some points. Plasma waves go harmlessly through all objects except asteroids. They have a maximum lifetime (you may choose the value) after which they dissipate and are removed from the world. See below for additional details.

Local Coordinates and Object Transformations

Previously, each object was defined and drawn directly in terms of its screen coordinates – that is, each object's `draw()` method used coordinates which were screen values. Now, each object should be drawn in its own *local coordinate system*, and should have a set of **AffineTransform** (**AT**) objects defining its *current transformation* (one **AT** each for translation, rotation, and scaling). This set of transformations specifies how the object is to be transformed from its local coordinate system into the world coordinate system (or into its *parent's* coordinate system in the case of hierarchical objects; see the "Fireball" example in the notes).

Previously, the `draw()` method for each moving object needed only to worry about drawing a simple object shape at the proper location. Now it also needs to apply the "current transformation" of the object so that the object will be properly drawn. This is done utilizing the mechanism discussed in class: the `draw()` method saves the current **Graphics2D** transform, appends the transformation of the object onto the **Graphics2D** transform, draws the object (and its sub-objects, in the case of a hierarchical object) using *local coordinate system* draw operations, and then restores the saved **Graphics2D** transform. That is, each `draw()` method temporarily adds its own object's local transformations to the **Graphics2D** transformation prior to invoking drawing operations, and then restores the **Graphics2D** transformation before returning.

All object drawing commands specify the object appearance in "local object coordinate space". That is, all drawing commands must be relative to the "local origin" (0,0). This is different from the previous assignment, where your `draw()` commands were relative to the "location" of the object. Now, the "location" is being set in the translation transformation added to the **Graphics2D** object prior to doing the actual drawing. (For example, if you previously had a draw command like `g.drawRect(xLoc,yLoc,width,height)`, it becomes `g.drawRect(0,0,width,height)`, drawing in "local space" at (0,0) – which is then translated by the AT to the proper location.)

In addition, hierarchical components which change dynamically should have the change applied via transformations. For example, movement of the hierarchical components of a missile as well the tumbling of asteroids (described earlier) should be applied via an AT.

World/Screen Coordinates

Your program must maintain a *Viewing Transformation Matrix* (VTM) which contains the series of transformations necessary to change world coordinates to screen coordinates. This VTM is then applied to every coordinate output during a repaint operation. The VTM is simply an instance of the Java **AffineTransform** class, named (for example) `theVTM`.

To apply the VTM during drawing, your `MapView` display container's `paint Component()` method should build an updated VTM and concatenate it into the **AffineTransform** of the

Graphics2D object used to perform the drawing. **Paint Component()** then passes this **Graphics2D** object to the **draw()** method of each shape. As described above, each **draw()** method will then in turn temporarily add its own object's local transformations to that same **Graphics2D** transformation.

In order to build a correct VTM, the program uses the appropriate translate and scale operations to map the world coordinates onto a (hypothetical) Normalized Device (ND) and then onto the screen, utilizing the current "world window" and "container size" values to build the VTM as described in the lecture notes.

Pointer Input

A pointer event contains a **Point** giving the current location *in screen coordinates*. However, when selecting asteroids and missiles (in pause mode), pointer input needs to determine *world* locations. Therefore, during selection the program must transform pointer input coordinates from screen units to world units. To do this, apply the *inverse* of the current VTM to the pointer coordinates (producing the corresponding point in the world). This world point can then be passed to the **contains()** method of each object, which converts the world point into local coordinates (by applying the inverse of the object's local transforms) and uses the result to determine whether the object contains the specified point.

Bezier Curves

Each time the 'P' key is pressed the ship should generate a *different* plasma wave (curve). Each new curve is to be defined by 4 *randomly-generated* control points. The range of the (x,y) values of the control points should be constrained such that the curve lies (more or less) "ahead of" the ship. The range should also be constrained such that the curve can be as much as about five times the size of the ship (but no more; otherwise a single plasma shot becomes a "doomsday" weapon). Note that since the control points are random (although constrained), some curves will be small while others will be large, and each will be a unique shape. Note also that the curve becomes a new world object, moving in the direction the ship was heading, and remains in the world until it collides with an asteroid or dissipates.

In addition to drawing the curve itself, the draw routine must also draw three straight lines connecting the control points (in order) so that the control polygon of the curve can easily be seen (this is to make it easier to verify that your Bezier curve-drawing routine is functioning properly; Bezier curves have the property that they always lie within their "control polygon"). You may choose the color scheme for drawing the curve and the control polygon lines. You may also add a command to hide the drawing of the control polygon lines, although it must be ON by default. As with other output, the drawing routines for the curve should use *local coordinates* to draw the curve and its outline, relying on the application of the local transform to map the output to world coordinates. Also, the drawing routine for the curve must use the recursive implementation (not an iterative implementation).

Note that the control points for a Bezier curve defining a given plasma wave do not change once the wave is created; the same control points are used in drawing the plasma wave for the lifetime of that wave. Note also that zooming in on a plasma wave should show the wave's curve in increasing detail (this is the reason for requiring the recursive, as opposed to iterative, drawing method).

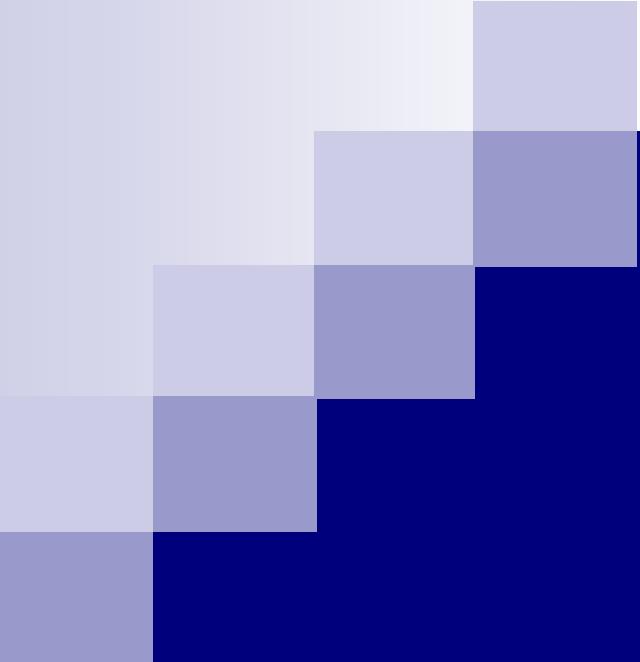
Additional Notes

- The blink rate of Space Stations is a value in *seconds*; if you did not adjust your Space Stations to blink on/off at the specified rate in A3 then you must make that correction for this assignment.

Deliverables

Submit a single .zip file to Canvas containing source code, compiled (.class), and resource files (e.g. sounds and image files). Your program must be in a single package named “**a4**”, with a main class named “**Starter**”.

The due date for this assignment is Friday, May 10th at midnight.



Assignment 1

Discussion

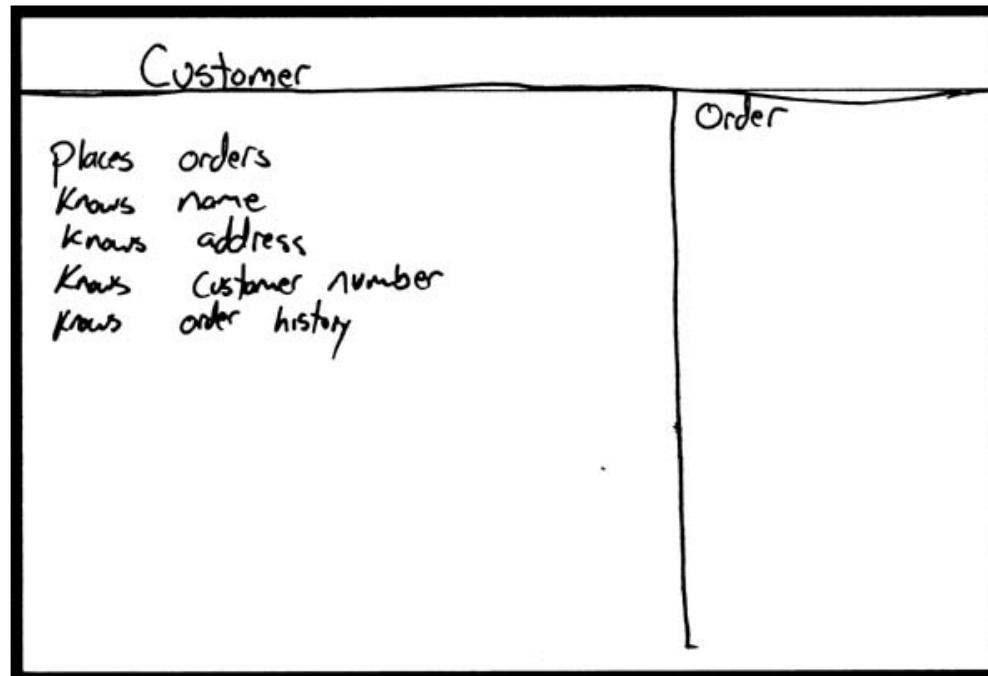
(Spring 2019)

Key Points

- Read the assignment MANY times.
 - See appendix at the end (please not skip it)
 - Sample code provided there.
- Analysis
 - Identify Suitable Classes
 - Use hints from assignment
 - Use CRC form: Class Responsibility Collaboration
 - Check your work again² your requirements

Class Name	
Responsibilities	Collaborators

Example of CRC card



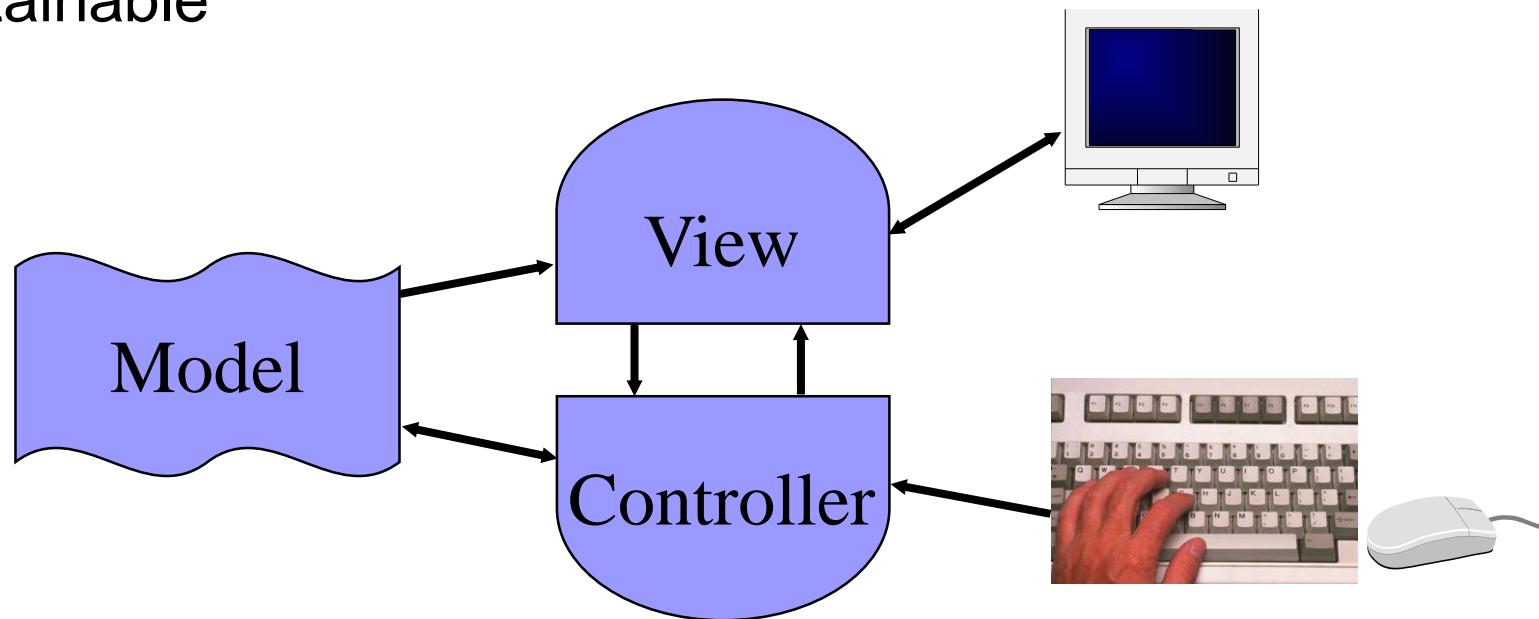
Professor	Name Address Phone number Email address Salary Provide information Seminars instructing	Seminar
Transcript	**See the prototype** Determine average mark	Student Seminar Professor Enrollment
Enrollment	Mark(s) received Average to date Final grade Student Seminar	Seminar
Student Schedule	**See the prototype**	Seminar Professor Student Enrollment Room
Room	Building Room number Type (Lab, class, ...) Number of Seats Get building name Provide available time slots	Building
Building	Building Name Rooms Provide name Provide list of available rooms for a given time period	Room

Source: <http://agilemodeling.com/artifacts/crcModel.htm>

Model-View-Controller

(Assignment 1 Game Structure)

- ◆ Partitions application into **Model**, **View**, **Controller** so that it is
 - scalable
 - maintainable



Model-View-Controller

design pattern (Cont)

Component	Purpose	Description
Model	Maintain data	Business logic plus one or more data sources such as a database.
View	Display all or a portion of the data	The user interface that displays information about the model to the user.
Controller	Handle events that affect the model or view	The flow-control mechanism means by which the user interacts with the application.

Model-View-Controller

design pattern (Cont)

Component	In our assignment # 1 context (Spring 2019)
Model (GameWorld)	A game in turn contains several components, including (1) a GameWorld which holds a collection of game objects and other state variables . Later, we will learn that a component such as GameWorld that holds the program's data is often called a model.
View (Future: Map and Score Views)	In this first version of the program the top-level Game class will also be responsible for displaying information about the state of the game. In future assignments we will learn about a separate kind of component called a view which will assume that responsibility.
Controller (Game)	The top-level Game class also manages the flow of control in the game (such a class is therefore sometimes called a controller). The controller enforces rules such as what <u>actions a player may take and what happens as a result</u> . This class accepts input in the form of keyboard commands from the human player and invokes appropriate methods in the game world to perform the requested commands – that is, to manipulate data in the game model.

Model-View-Controller design pattern (Cont)

CN1 Starter Class – Just add new Game()

Model
(GameWorld)

```
class Starter {  
    //other methods  
    public void start() {  
        if(current != null){  
            current.show();  
            return;  
        }  
        new Game();  
    }  
    //other methods  
}  
  
public class GameWorld {  
    public void init(){  
        //code here to create the  
        //initial game objects/setup  
    }  
    // additional methods here to  
    // manipulate world objects and  
    // related game state data  
}
```

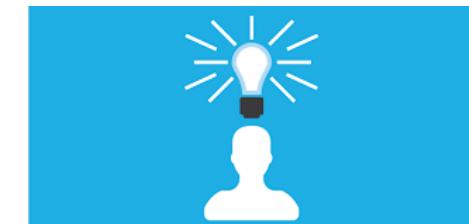
Controller
(Game)

```
import com.codename1.ui.Form;  
public class Game extends Form{  
    private GameWorld gw;  
  
    public Game() {  
        gw = new GameWorld();  
        gw.init();  
        play();  
    }  
  
    private void play() {  
        // code here to accept and  
        // execute user commands that  
        // operate on the game world  
        // (refer to "Appendix - CN1  
        // Notes" for accepting  
        // keyboard commands via a text  
        // field located on the form)  
    }  
}
```

View (in A1 only)
(Game – Will
be having a separate
Components in A2)

Controller: Process input commands (from A1 Appendix)

```
import com.codename1.ui.events.ActionListener;
import com.codename1.ui.Label;
import com.codename1.ui.TextField;
import com.codename1.ui.events.ActionEvent;
import java.lang.String;
private void play()
{
    Label myLabel=new Label("Enter a
Command:");
    this.addComponent(myLabel);
    final TextField myTextField=new TextField();
    this.addComponent(myTextField);
    this.show();
    myTextField.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent evt) {
            String sCommand=myTextField.getText().toString();
            myTextField.clear();
            switch (sCommand.charAt(0)){
                case 'e':
                    gw.eliminate();
                    break;
                //add code to handle rest of the commands
            } //switch
        } //actionPerformed
    } //new ActionListener()
); //addActionListener()
} //play
```



```
switch (sCommand.charAt(0)) {
    case 'a':
        gw.addAsteroid();
        break;
```

End of Phase 1: Do Design

Work & Drawing a Sketch, Code & Test



- Do a short design work
- Draw a UML sketch of the current classes (Starter, Game, GameWorld)
- Code it
- Run Test Case # 1 (Can you pass this test ?)
- Refactor



Model: GameWorld Process

Add Asteroid Command

```
public class GameWorld {  
    Random random = new Random();  
    public Vector<GameObject> store = new Vector<GameObject>();  
  
    public void addNewAsteroid() {  
        //Create an Asteroid object  
        Asteroid asteroid = new Asteroid();  
        //Add Asteroid to storage vector  
        store.add(asteroid);  
        //Tell user you created an Asteroid  
        System.out.println("A new ASTEROID has been created.");  
    }  
}
```

And others command

Asteroid Concrete Class

(Sample Only)

```
public class Asteroid extends MovableGameObject {
    private int size;

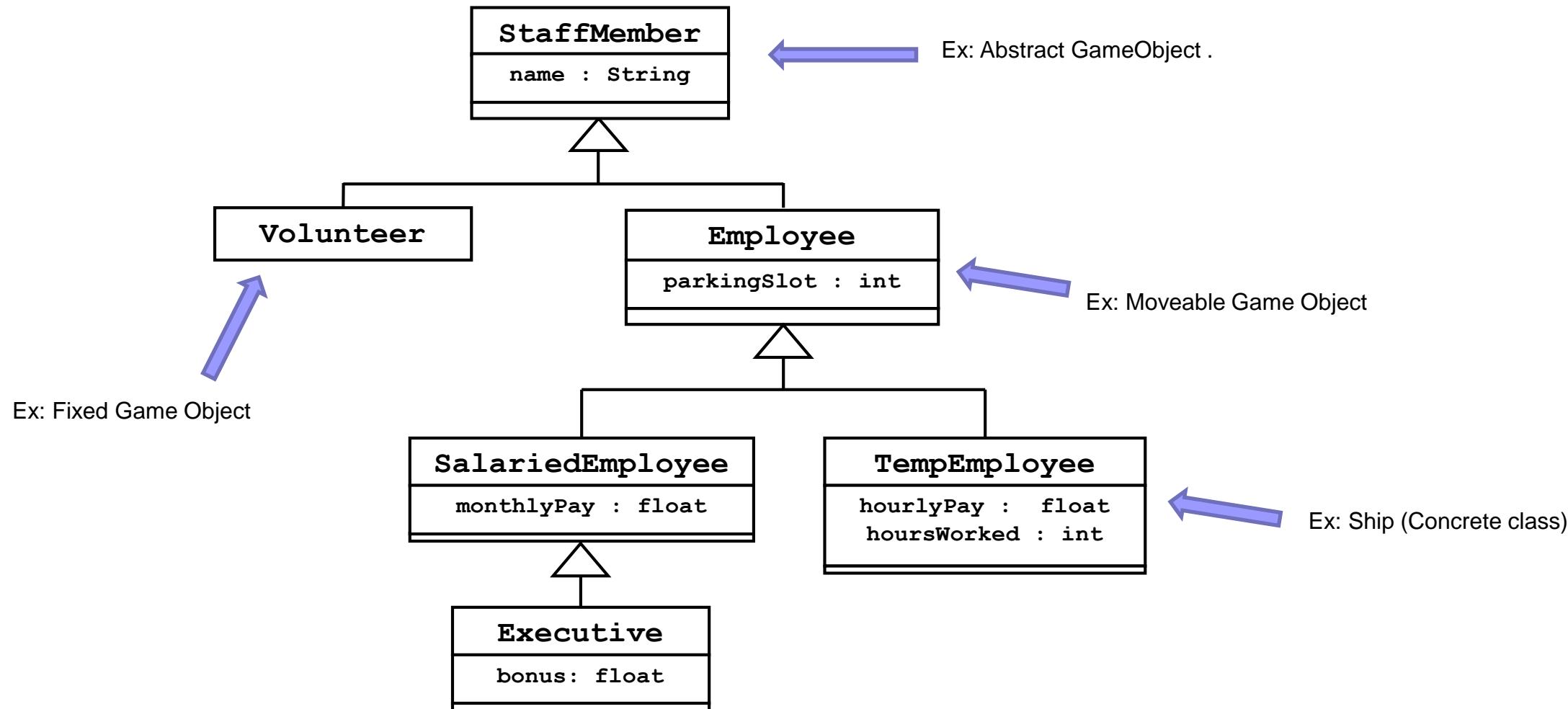
    public Asteroid() {
        super(ColorUtil.BLACK);
        final int MIN_SIZE = 6;
        final int MAX_SIZE = 30;
        this.size = GameObject.rand.nextInt(MAX_SIZE - MIN_SIZE + 1) + MIN_SIZE;
    }

    public int getSize() {
        return this.size;
    }

    @Override
    public String toString() {
        return (
            "Asteroid: loc=" + GameObject.round(getX()) + "," + GameObject.round(getY()) +
            " color=" + GameObject.getColorString(getColor()) +
            " speed=" + GameObject.round(getSpeed()) +
            " dir=" + getDirection() +
            " size=" + this.getSize()
        );
    }
}
```

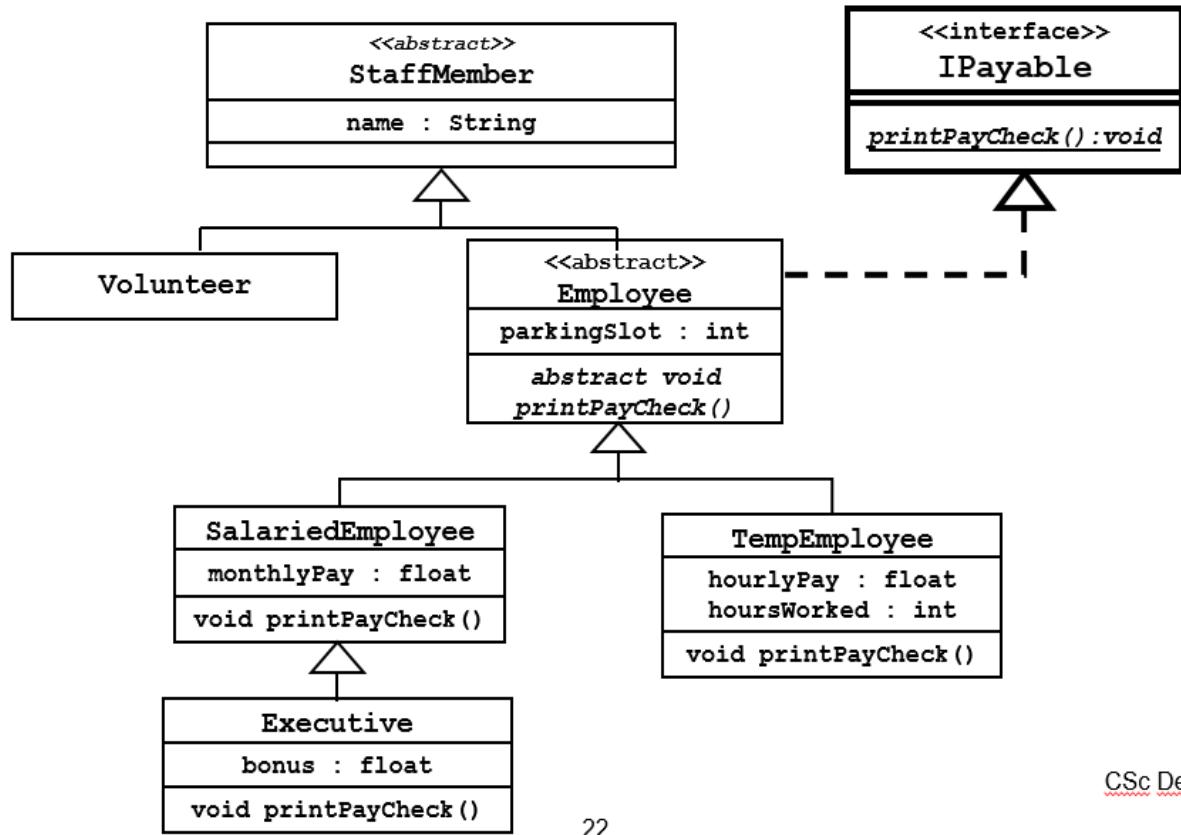
Now, add others commands and their objects

Now, think sub-classes (Connecting to our Lecture Materials)



Think sub-classes (Cont)

- StaffMember hierarchy using Interfaces:



Can your work support Runtime Polymorphism safely?

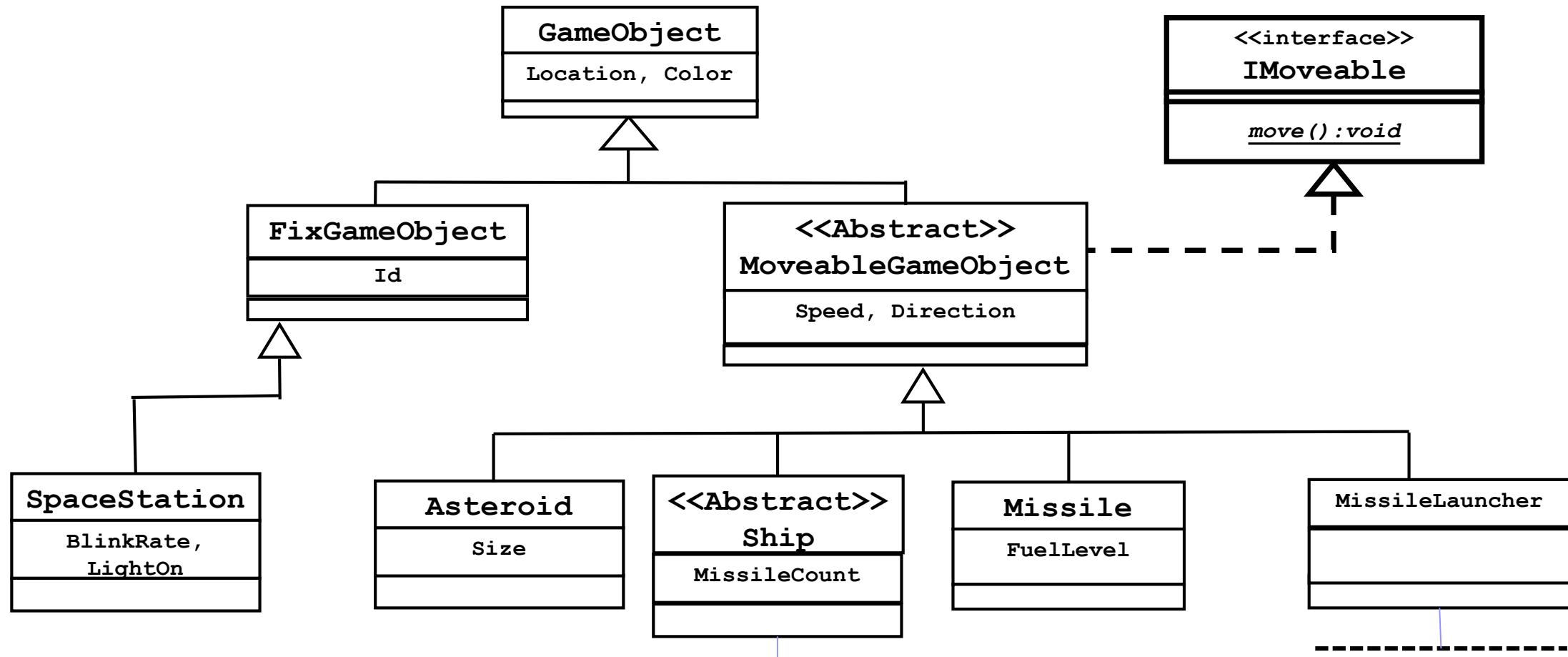
```

for (int i=0; i<theWorldVector.size(); i++) {
    if (theWorldVector.elementAt(i) instanceof IMovable) {
        IMovable mObj = (IMovable)theWorldVector.elementAt(i);
        mObj.move();
    }
}
  
```

Note: See lecture on Polymorphism
And “Additional details” note in Assignment 1.

CSc Dept, CSUS

Think sub-classes – Example Only (Not Complete)



End of Phase 2: Do Design

Work & Drawing a Sketch, Code & Test

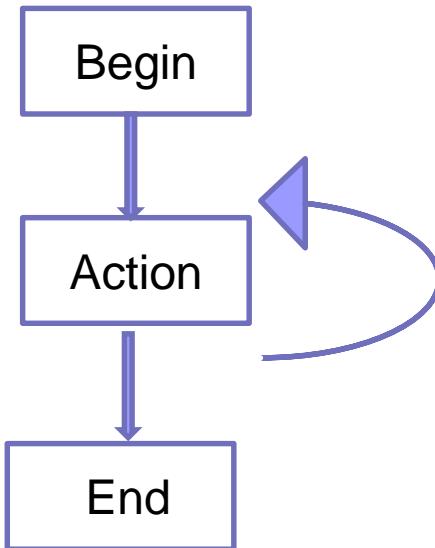


- Do a short design work
- Draw a UML sketch of the current classes (Asteroid, Missile, Station, PS, NPS, Launchers..)
- Code each class
- Run Test Case # 2 (Can you pass this test ?)
 - Check output format and correct state for each object
- Refactor



Start Phase 3 and 4: Do Design

Work & Drawing a Sketch, Code & Test



- Do a short design work
- Implement **animation functions** (i.e. 't', 'b', 'r', '>')
- Implement **collision functions** (i.e. 'k', 'e', 'E', 'c' ...)
- Code each function
- Run Test Case # 3 (Can you pass this test ?)
 - Check output format and correct state for each object by running both 'm' and 'p' commands.
- Refactor

-
- Do a short design work
 - Implement game end **function** (i.e. 'q', or player ran out of lives)
 - Code the function
 - Run test case # 4 (Can you pass this test ?)
 - Refactor

Additional information

(Do not forget these in your UML Diagram!)

- Including Interfaces (2)
- Declare Methods, Attributes, Modifier (i.e. Private)
- Include external packages and class names (reference in UML diagram)
 - com.codename1.ui.Form
- Other classes: Starter, Game, GameWorld (Do not forget – See slide 7)
- Other relationships: Composition, Dependency, Association.
- Concrete classes have to `toString` method (see Asteroid Slide 11)
- Use Codename one provided **Point** or **Point2D** to hold x,y instead of int x,int y –
For object location (See assignment 1 note)

Coding after completion of UML Diagram

- Expect to return to UML diagram to make changes once identified issues identifying through coding or testing
- Including comments and correct use of variables, class names, package names
- Proper use of inheritance, encapsulation, polymorphism, interface

Other Information

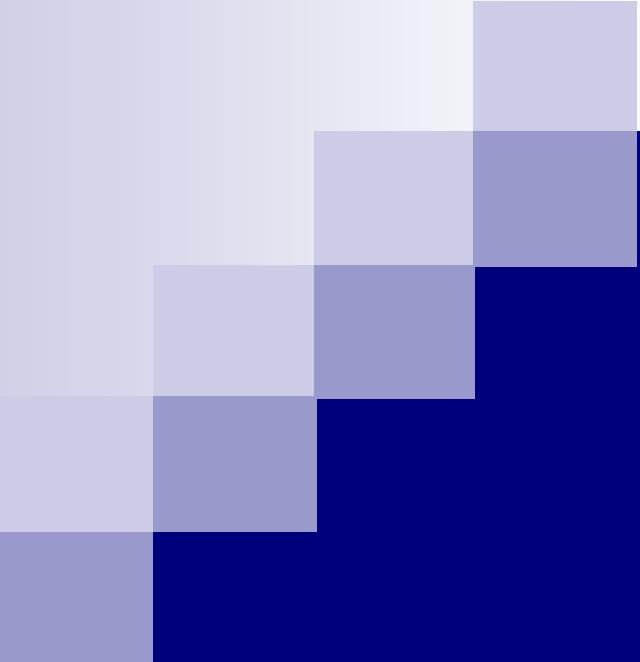
- Can I code the entire A1 without UML diagram? Answer is no.
- Can I automatically generate UML diagram from my Code (code/fix) ? Answer is no.
- Prototyping some key concepts ? Yes, I recommend it.
 - I.e. Input processing ?
 - Calling the game object?

Other Information

- Validate the complete set of commands
- Check for input errors and boundary conditions
 - Your program should not be crashed under these conditions (software quality). Output error handling text when required.
 - Can your program handle this as input: !@\$%^Aa ?
 - Conform to expected output format (i.e. decimal points)

Turning the A1 assignment (Suggestions)

- Check the deliverable section of the assignment
- Refresh the dist folder ← (Watch out for old code: No “Hello World” or A0)
- Run the launch command to ensure the program can launch correctly
- Turn in your work before Feb **25th before 11:59 PM**
- Have fun ☺ but not procrastinated



Assignment 2 Discussion

Starter Kit

(3/14/19)

Assignment 2

Two main focuses for A2:

- Graphical User Interface (GUI)
- To incorporate several important *design patterns*

The rest of the game will appear to the user to be similar to the one in A1, and most of the code from A1 will be reused, although it will require some modification and reorganization.

Assignment 2 GUI

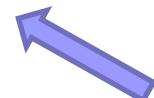


Transform

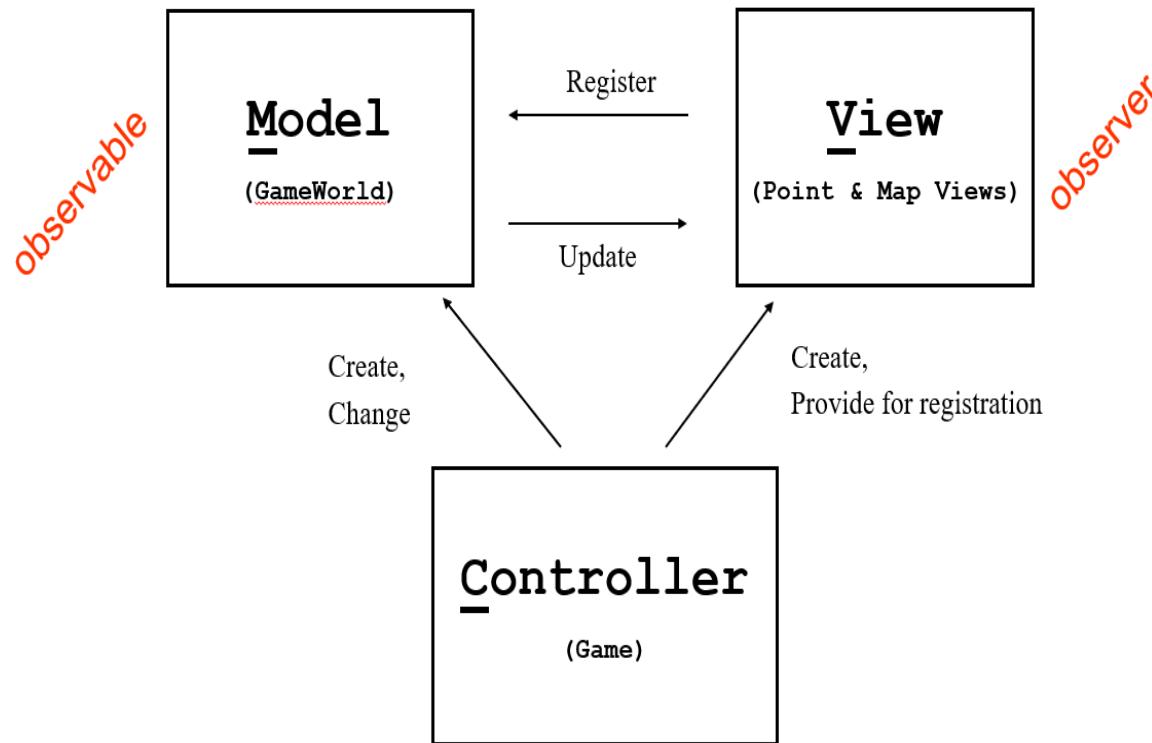


(Note: This is a mockup screen only. Not all the commands are displayed here)

Note: To get started,
please see
ContainerDemo in Canvas
(SampleUIProgram)
for practice.



Assignment 2 – MVC Architectural Pattern



The **Game** class containing the **getCommand()** method along with methods to process the various commands (some of which call methods in **GameWorld** when access to the game objects is needed).

The **GameWorld** class contains a collection of game objects and other game state values (more details below).

You are also required to add two classes acting as “views”: a *Point* view which will be graphical, and a *Map* view which will retain the text-based form generated by the ‘m’ command in A1

Game Class High Level Steps

- Create “**Observable**” - gw = new GameWorld();
- Create an “**Observer**” for the map and create an “**Observer**” for the points
 - mv = new MapView() and pv = new PointsView(gw)
- Register observers: gw.addObserver(mv)
gw.addObserver(pv);
- And more below:

```
// code here to create menus, create Command objects for each command,  
// add commands to Command menu, create a control panel for the buttons,  
// add buttons to the control panel, add commands to the buttons, and  
// add control panel, MapView panel, and PointsView panel to the form
```

(from assignment 2)

```
// Adding PointsView and MapView to the Form  
add(BorderLayout.NORTH, pv);  
add(BorderLayout.CENTER, mv);
```

IMPORTANT!

Notes: (1) I will use PointView and ScoreView interchanged in this lecture
(2) Use BorderLayout for your containers

GameWorld Class High Level Steps

(from assignment 2)

```
public class GameWorld extends Observable implements IGameWorld {  
    // code here to hold and manipulate world objects, handle observer registration,  
    // invoke observer callbacks by passing a GameWorld proxy, etc.  
}
```

Implements IGameWorld

(To be discussed in Proxy
Design Pattern)

(To be discussed in
Iterative Design Pattern)

```
public class GameWorld extends Observable {  
    //~~~~~ F I E L D S ~~~~~/  
    //~~~~~ C O N S T R U C T O R S ~~~~~/  
  
    //All state variables are stored here  
    private GameCollection go;  
  
    private boolean soundOn;  
  
    /* This constructor sets all state values appropriately. This also creates the  
     * collection necessary to hold all game objects.  
     */  
    public GameWorld(){  
        go = new GameCollection();  
        this.init(); //Then initialize the world  
    }  
}
```

(More on: Callbacks by passing GameWorld Proxy Later)

PointsView Class High Level Steps

(from assignment 2)

```
public class PointsView extends Container implements Observer {  
    public void update (Observable o, Object arg) {  
        // code here to update labels from data in the Observable (a GameWorldPROXY)  
    }  
}  
  
public class PointsView extends Container implements Observer  
{  
    private Label pointsValueLabel;  
    public PointsView()  
    {  
        // Instantiate text labels  
        Label pointsTextLabel = new Label("Points:");  
        // Instantiating value labels  
        pointsValueLabel = new Label("XXX");  
        // Set color  
        pointsTextLabel.getAllStyles().setFgColor(ColorUtil.rgb(0,0,255));  
  
        // Adding a container with a boxlayout  
        Container myContainer = new Container();  
        myContainer.setLayout(new BoxLayout(BoxLayout.X_AXIS));  
  
        // Adding all labels in order  
        myContainer.add(pointsTextLabel);  
        this.add(myContainer);  
    }  
}
```

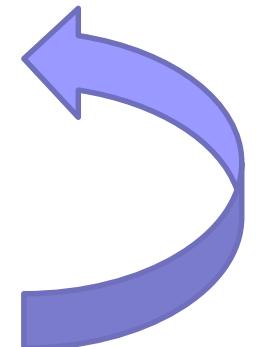
Showing only
One Attribute

PointsView Class High Level Steps (Cont)

(from assignment 2)

```
public class PointsView extends Container implements Observer {  
    public void update (Observable o, Object arg) {  
        // code here to update labels from data in the Observable (a GameWorldPROXY)  
    }  
  
    @Override  
    public void update(Observable observable, Object data) {  
        IGameWorld gw = (IGameWorld) data; ←  
        this.pointsValueLabel.setText("'" + gw.getPlayerScore() );  
        // ....  
        this.repaint();  
    }  
}
```

Showing only
One Attribute



Note: GameWorld PROXY be casted to type IGameWorld in order to access the GameWorld methods in it!

MapView Class High Level Steps

(from assignment 2)

- a **MapView** class containing code to output the map.
- The **MapView** output for this assignment is unchanged from A1: text output on the console.
- Code update method as in PointView to output text on the console.
- Iterate through the collection via Iterative Design Pattern.

Assignment 2 – Design Patterns

The program must use appropriate interfaces for organizing the required design patterns.

In all, the following design patterns are to be implemented in this assignment:

- *Observer/Observable* – to coordinate changes in the data with the various views,
- *Iterator* – to walk through all the game objects when necessary,
- *Command* – to encapsulate the various commands the player can invoke,
- *Proxy* – to insure that *views* cannot *modify* the game world.

Command Design Pattern

(from assignment 2)

The approach you must use for implementing command classes is to have each command extend the CN1 build-in Command class (which implements the ActionListener interface), as shown in the course notes. Code to perform the command operation then goes in the command's actionPerformed() method. Hence, actionPerformed() method of each command class that performs an operation invoked by a single-character command in A1, should call the appropriate method in the GameWorld that you have implemented in A1 when related single-character command is entered from the text field (e.g., accelerate command's actionPerformed() would call accelerate() method in GameWorld).

Extend from
build-in
command
class



Calling appropriate
Method in
GameWorld

Example: Command Design Pattern

```
6 public class AddAsteroidCommand extends Command {  
7     //~~~~~ F I E L D S ~~~~~//  
8     //~~~~~ C O N S T R U C T O R S ~~~~~//  
9     //~~~~~ M E T H O D S ~~~~~//  
10  
11     private GameWorld gw;    //Reference to a Game World  
12     //~~~~~ C O N S T R U C T O R S ~~~~~//  
13     //~~~~~ M E T H O D S ~~~~~//  
14     /* There is only one constructor.  
     */  
15     public AddAsteroidCommand( GameWorld gw ){  
16         super( "Add Asteroid" );  
17         this.gw = gw;  
18     }  
19     //~~~~~ M E T H O D S ~~~~~//  
20     //There is only one method to override the action performed  
21     @Override  
22     public void actionPerformed( ActionEvent e ){  
23         gw.addAsteroid();  
24         System.out.println("Add Asteroid.");  
25     }  
26 }  
27  
28  
29  
30  
31 }
```

(Note: assignment 1)

Example: Command Design Pattern (Cont)

Button class is automatically able to be a “holder” for command objects; Button have a setCommand() method which allows inserting a command object into the button. Command automatically becomes a listener when added to a Button via setCommand() (you do not need to also call addActionListener()), and the specified Command is automatically invoked when the button is pressed, so if you use the CN1 facilities correctly then this particular observer/observable relationship is taken care of automatically.

```
//Add the new buttons that will be on the west border
Button addAsteroid = new Button ( "Add Asteroid" );

//Make all buttons look prettier
//Cyan 'Asteroid'
addAsteroid.getAllStyles().setBgTransparency( 255 );
addAsteroid.getUnselectedStyle().setBgColor( ColorUtil.rgb( 0, 150, 150 ) );
addAsteroid.getAllStyles().setFgColor( ColorUtil.rgb( 255, 255, 255 ) );

//Set some button padding
addAsteroid.getAllStyles().setPadding( TOP, 5 );
addAsteroid.getAllStyles().setPadding( BOTTOM, 5 );
```

Suggest:

In Game Constructor

Example: Command Design Pattern

(Cont)

The Game constructor should create a single instance of each command object (for example, a “Accelerate” command object, etc.), then insert the command objects into the command holders (buttons, side menu items, title bar area items) using methods like setCommand() (for buttons), addCommandToSideMenu() (for side menu items), and addCommandToRightBar() (for title bar area items). You should also bind these command objects to the keys using addKeyListener() method of Form. You must call

```
// ~~~~~ ALL COMMANDS BELOW ~~~~~  
  
//Declare all the needed commands for the buttons, keys, and side bar  
AddAsteroidCommand myAddAsteroid = new AddAsteroidCommand(gw);  
  
addAsteroid.setCommand( myAddAsteroid );  
  
//Then the commands to the keys  
addKeyListener( 'a', myAddAsteroid );
```



Demonstration code!

Example: Command Design Pattern

Arrows and Space bar (Cont)

The second input mechanism will use CN1 *Key Binding* concepts so that the *left arrow*, *right arrow*, *up arrow*, and *down arrow* keys invoke command objects corresponding to the code previously executed when the “l”, “r”, “i”, and “d” keys (for changing the ship direction and speed) were entered, respectively. Note that this means that whenever an arrow key is pressed, the program will *immediately* invoke the corresponding action (no “Enter” key press is required). The program is also to use key bindings to bind the SPACE bar to the “fire missile” command and the “j” key to the “jump through hyperspace” command. If you want, you may also use key bindings to map any of the other command keys from A1, but only the ones listed above are required.

```
//===== Binding play-mode specific key commands =====
// Binding up arrow to increase ship speed (up arrow key code = -91)
addKeyListener(-91, myIncreaseShipSpeedCmd);
// Binding down arrow to decrease ship speed (down arrow key code = 92)
addKeyListener(-92, myDecreaseShipSpeedCmd);
// Binding left arrow to turn the ship left (left arrow key code = -93)
addKeyListener(-93, myTurnShipLeftCmd);
// Binding right arrow to turn the ship right (right arrow key code = -94)
addKeyListener(-94, myTurnShipRightCmd);
// Binding the space bar to fire a ship missile (space bar key code = -90) (this also binds the enter key)
addKeyListener(-90, myFireShipMissileCmd);
// ... . . .

addKeyListener(44, mGCmdl); // MissileLauncher Left turn
addKeyListener(46, mGCmdr); // MissileLauncher Right turn
```

Iterative Design Pattern

(from assignment 2)

Iterator Design Pattern

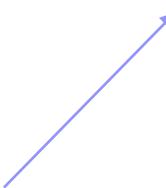
The game object collection must be accessed through an appropriate implementation of the Iterator design pattern. That is, any routine that needs to process the objects in the collection must not access the collection directly, but must instead acquire an iterator on the collection and use that iterator to access the objects. You should develop your own interfaces to implement this design pattern, instead of using the related build-in CN1 interfaces. The game object collection will implement an interface called **ICollection** which defines at least two methods: for adding an object to the collection (i.e., **add()**) and for obtaining an iterator over the collection (i.e., **getIterator()**). The iterator should exist as a private inner class inside game object collection class and should implement an interface called **IIterator** which defines at least two methods: for checking whether there are more elements to be processed in the collection (i.e., **hasNext()**) and returning the next element to be processed from the collection (i.e., **getNext()**).

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

SpaceCollection With Iterator

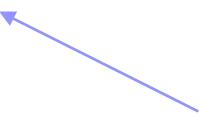
Your Game Collection



```
/** 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 IIIterator getIterator() {  
        return new SpaceVectorIterator ( ) ;  
    }  
  
    ...continued...
```

Your Game ICollection



SpaceCollection With Iterator (cont.)

Inner Class Iterator

```
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
```

Example: Using An Iterative Design Pattern

```
//This will update the map eventually. For now it displays all the game objects
public void update( Observable o, Object arg ){
    System.out.println( "Map Width: " + Game.getMapHeight() + " Map Height: "
        + Game.getMapWidth() );
    //Cast the Observable objects as the GameWorld first to access variables
    GameWorld gw = (GameWorld)o;
    GameCollection go = gw.getGameObjects();
    IIterator gameIterator = go.getIterator();
    while ( gameIterator.hasNext() ){
        System.out.println( gameIterator.getNext() );
    }
}
```



Note: This slide does not use Proxy Gameworld object – Use arg instead of o
See slides on Proxy Design Pattern next.

Use: arg Proxy object for your A2 work.

Proxy Design Pattern

(from assignment 2)

Proxy Design Pattern

To prevent views from being able to modify the `GameWorld` object received by their `update()` methods, the model (`GameWorld`) should pass a `GameWorld proxy` to the views; this proxy should allow each view to obtain all the required data from the model while prohibiting any attempt by the view to *change* the model. The simplest way to do this is to define an interface `IGameWorld` listing the methods provided by a `GameWorld`, and to provide a new class `GameWorldProxy` which implements this same interface. The `GameWorld` model then passes to each observer's `update()` method a `GameWorldProxy` object instead of the actual `GameWorld` object. See the attachment at the end for additional details.

Recall that there are two approaches which can be used to implement the Observer pattern: defining your own `IObservable` interface, or extending the build-in CN1 Observable class. You are required to use the latter approach (where your `GameWorld` class extends `java.util.Observable`). Note that you are also required to use the build-in CN1 Observer interface (which also resides in `java.util` package).

Proxy Design Pattern (Cont)

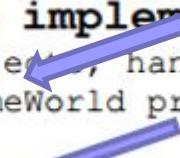
IGameWorld Interface

```
public interface IGameWorld {  
    //specifications here for all GameWorld methods  
}  
  
4  
5 public interface IGameWorld  
6 {  
7     // Returns the player score of the GameWorld  
8     int getPlayerScore();  
9     // Get other games attributes |.....
```

Proxy Design Pattern (Cont)

notifyObservers - Whenever you need to update the Views

```
public class GameWorld extends Observable implements IGameWorld {  
    // code here to hold and manipulate world objects, handle observer registration,  
    // invoke observer callbacks by passing a GameWorld proxy, etc.  
}  
  
    gameObjects.add(asteroid);  
  
    System.out.println("A new asteroid has been created.");  
  
    this.setChanged();  
    this.notifyObservers(new GameWorldProxy(this));
```



Proxy Design Pattern

(from assignment 2)

Remember Keyword: **forward!**

```
public class GameWorldProxy extends Observable implements IGameWorld {
    // code here to accept and hold a GameWorld, providing implementations
    // of all the public methods in a GameWorld, forward allowed
    // calls to the actual GameWorld, and reject calls to methods
    // which the outside should not be able to access in the GameWorld.
}

8 public class GameWorldProxy extends Observable implements IGameWorld
9 {
10     private GameWorld gw;
11
12⊕     public GameWorldProxy(GameWorld gw)
13     {
14         this.gw = gw;
15     }
16
17     // Returns the player score of the GameWorld
18⊕     public int getPlayerScore()
19     {
20         return gw.getPlayerScore();
21     }
22 }
```

Proxy Design Pattern (Cont)

(Inside Update – Observer – PointView)

(Use Data passed over from Observable)

```
Showing only  
One Attribute  
@Override  
public void update(Observable observable, Object data) {  
    IGameWorld gw = (IGameWorld) data;  
    this.pointsValueLabel.setText(" " + gw.getPlayerScore());  
    // ....  
    this.repaint();  
}
```

Note: GameWorld PROXY be casted to type IGameWorld in order to access the GameWorld methods in it!

Others Important Information

- A2 assignment is due on 4/1st.
 - 5 % deduction rule enforced strictly by the Graders.
- Be sure to be able to run the command line successfully - launching of a2 (using the same convention discussed in the Course module 2)
 - Check for up-to-date jar file (non zero size!)
 - Test cases first delivery due on 3/14 with their test results due on 4/1st . **Lateness count.**
 - **No regrading on A2.**
- A1's problematic functional areas including design and coding (areas were not implemented or having issues reported by Graders), if not addressed, will be deducted again on A2.
 - If addressed successfully, in a readme.txt, a grader will recommend to the instructor for an improvement score on A1 (5-10 points)
- Read your A2 several times.
 - If there is anything not clear, please check with your instructor.

A2 Friendly Reminder Messages From your Graders

1. Ensure that the game knows the **difference between a PS missile and NPS** missile that way the collision methods know the correct missile involved.
2. Properly test the **move** function as many students did not perform the correct computations. Use Radiant for Math function (see A2 discussion)
3. Inform Mac users to make sure they name their package correctly otherwise it will not work through command line. Follow lecture notes.
4. Deliverable does not including their src program or UML, all of which **cost the students a lot of points**. **CHECK** your deliverables section in A2.
5. Not implementing all the commands like increase/decrease speed or turn right/left. Even with the deliverable, it might be good to remind the students to **start their assignments early and carefully read the instructions**.
6. Several students had packages named something other than a1 which caused me to be unable to run their programs, so it may be beneficial to remind the students to carefully read the setup instructions and test their program with the RunAssignment.jar.

Attendance Quiz # 1 Results for Dennis Dang

Score for this attempt: **10** out of 10

Submitted Jan 24 at 6:08pm

This attempt took 1 minute.

Question 1

10

/ 10 pts

1. Name the states of a life cycle of a Codename one application. Explain how would you arrive at each state?

2. Name a program your instructor provided for you. This program is to confirm if you have the right configuration before submitting your assignment for grading. How would you execute this program?

Your Answer:

1. The three states of the Codename One application's life cycle are "Not Running", "Foreground", & "Suspended".

To be in the Foreground state:
the init() must be executed beforehand
the app resumes after being paused (was in "suspended" state). (The user clicks Simulate>Resume App)

To be in the Not Running state:
the app did not start at all by the user.

the app was terminated by the user; destroy() was called.
the app crashed

To be in the Suspended state:
the app was minimized by the user; stop() was called
To minimize the app via simulation, click on
Simulate>Pause App

2. The name of the program provided is
RunAssignment.jar

I may launch this via the command line by:
• copy RunAssignment.jar to the Project Folder
• Open up command line application
• type in: java -jar RunAssignment.jar A~~X~~Prj.jar
Where X = 0, 1, 2, 3, or 4.. depending on which project
version it is.

CSC-133 (Spring 2019)

Attendance Quiz 2 – OOP Concepts and UML Class Diagram

Student Name: _____ Key _____

Question 1: Two different programming teams have implemented a class named **Rectangle**. One team provided accessors to get and set the location (origin), width, and height of a rectangle, while the other team chose to make the origin, width, and height fields public so that they can simply be directly accessed (read and/or changed). The second team argues that if you have accessors which allow you to both get and set all the values in the rectangle, there is no difference in having the fields public. Explain why the second team does not know what they are talking about. Be specific; give an example of how their approach can produce a software system that fails. (10 points).

You have to have getters and setters for your code. You do not want anyone to have access to a class data members because they shouldn't easily be changed. A getter will only return the values, and a setter will check to make sure the new value is valid, rather than changing it anywhere, where it may be invalid.

An example would be:

A client, after a creation a Rectangle, he/she might attempt to change a Width of this Rectangle to a negative value. Then, then the area computing function will now return a negative value which is not correct. For example:

```
Rectangle myRec = new Rectangle(10, 20);
myRec.width = -2;
S.o.println("myRec area = " + myRec.getArea());
```

Question 2: UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. Please review the following slides. In the class diagram in slide 24, given the labels 1, 2, 3, 4, and 5 please fill out the required information in the table beneath. **BE SURE TO JUSTIFY YOUR ANSWER. (10 points).**

CSC 133 Lecture Notes
3 - OOP Concepts

Implementing Associations

```

classDiagram
    class Point
    class MainPanel
    class DisplayPanel

    Point "1" * --> MainPanel
    MainPanel "2" *--> DisplayPanel
    MainPanel "3" --> Point
    MainPanel "4" --> Point
    Point "5" --> DisplayPanel

    classDiagram
    class Point
    class MainPanel
    class DisplayPanel

    Point "1" * --> MainPanel
    MainPanel "2" *--> DisplayPanel
    MainPanel "3" --> Point
    MainPanel "4" --> Point
    Point "5" --> DisplayPanel
  
```

24 CSc Dept, CSUS

CSC 133 Lecture Notes
3 - OOP Concepts

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);
            }
        }
    }
  

```

26 CSc Dept, CSUS

CSC 133 Lecture Notes
3 - OOP Concepts

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);
        }
    }
  

```

25 CSc Dept, CSUS

CSC 133 Lecture Notes
3 - OOP Concepts

Class Point

- 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.
    }
  

```

27 CSc Dept, CSUS

Label Number	Name the association	Justifications	Specify Java Class(es) Name and Line of codes
①	Aggregation	Main Panel has myPoints (has relation). Points p is pre-existed outside.	<pre>public class MainPanel { private ArrayList<Point> myPoints; }</pre>
②	Composition	Main Panel has Display Panel. Create internal in Constructor. And not shared with the outside word.	<pre>public class MainPanel -class private DisplayPanel myDisplayPanel; public MainPanel() { myDisplayPanel = new DisplayPanel(this); }</pre>
③	Dependency	MainPanel uses DisplayPanel. DisplayPanel uses MainPanel. Source class has a reference to the dependent class	<pre>MainPanel: myDisplayPanel = new DisplayPanel(this); DisplayPanel: //establish linkage to my MainPanel myMainPanel = m ;</pre>
④	Aggregation	DisplayPanel has a field of MainPanel.	<pre>public class DisplayPanel private MainPanel myMainPanel;</pre>
5	Association	instances of DisplayPanel call method of MainPanel	<pre>DisplayPanel: //get the points from the MainPanel ArrayList thePoints = myMainPanel.getPoints();</pre>

CSC-133 (Spring 2019)
Attendance Quiz 2 – OOP Concepts and UML Class Diagram

Student Name: _____

Question 1: Two different programming teams have implemented a class named **Rectangle**. One team provided accessors to get and set the location (origin), width, and height of a rectangle, while the other team chose to make the origin, width, and height fields public so that they can simply be directly accessed (read and/or changed). The second team argues that if you have accessors which allow you to both get and set all the values in the rectangle, there is no difference in having the fields public. Explain why the second team does not know what they are talking about. Be specific; give an example of how their approach can produce a software system that fails. **(10 points)**.

Question 2: UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. Please review the following slides. In the class diagram in slide 24, given the labels 1, 2, 3, 4, and 5 please fill out the required information in the table beneath. **BE SURE TO JUSTIFY YOUR ANSWER. (10 points).**

CSC 133 Lecture Notes
3 - OOP Concepts

Implementing Associations

```

classDiagram
    class Point
    class MainPanel
    class DisplayPanel

    Point "1" * --> MainPanel
    MainPanel "2" *--> DisplayPanel
    MainPanel "3" --> DisplayPanel
    DisplayPanel "4" --> MainPanel
    DisplayPanel "5" --> MainPanel

    MainPanel "3" --> MainPanel
    MainPanel "4" --> DisplayPanel
    DisplayPanel "5" --> MainPanel
  
```

24 CSC Dept, CSUS

CSC 133 Lecture Notes
3 - OOP Concepts

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);
            }
        }
    }
  

```

26 CSC Dept, CSUS

CSC 133 Lecture Notes
3 - OOP Concepts

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);
        }
    }
  

```

25 CSC Dept, CSUS

CSC 133 Lecture Notes
3 - OOP Concepts

Class Point

- 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.
    }
  

```

27 CSC Dept, CSUS

Label Number	Name the association	Justifications (Please use the "Recap 1" Table (Slide # 33) for reference)	Specify Java Class(es) Name and Line of codes
①			
②			
③			
④			
⑤			

Attendance Quiz # 3 Results for Sidlak Caleb Malaki

Submitted Feb 7 at 4:25pm

This attempt took 12 minutes.

Question 1

Name the 4 basis of Polymorphism. Explain each.

Your Answer:

The 4 basis of polymorphism are inheritance, method overriding, polymorphic methods, and polymorphic assignment. Inheritance is when a subclass is derived from an existing superclass. Method overriding is when the subclass redefines the behavior of a method in its superclass. Polymorphic methods are methods that can take several forms. Polymorphic assignment is when a subclass object is assigned to a superclass variable.

CSC-133 (Spring 2019)
Attendance Quiz # 4

Student Name: _____ Key _____

Inheritance and Polymorphism (20 points). Consider the following classes. In the table below, indicate in the right-hand column the output produced by the statement in the left-hand column. If the statement produces more than one line of output, indicate the line breaks with slashes as in "a / b / c" to indicate three lines of output with "a" followed by "b" followed by "c". If the statement causes an error, fill in the right-hand column with the phrase "error" to indicate this.

“S.o.pln” means “System.out.println”.

Statement	Output
public class Eye extends Mouth { public void method1() { S.o.pln("Eye 1"); super.method1(); } } public class Mouth { public void method1() {	var1.method1(); Nose 1 var2.method1(); Eye 1/Mouth 1 var3.method1(); Eye 1/Mouth 1

<pre> S.o.pln("Mouth 1"); public void method2() { S.o.pln("Mouth 2"); method1(); } } public class Nose extends Eye { public void method1() { S.o.pln("Nose 1"); } public void method3() { S.o.pln("Nose 3"); } } public class Ear extends Eye { public void method2() { S.o.pln("Ear 2"); } public void method3() { S.o.pln("Ear 3"); } } The following variables are defined: Mouth var1 = new Nose(); Ear var2 = new Ear(); Mouth var3 = new Eye(); Object var4 = new Mouth(); Eye var5 = new Nose(); Mouth var6 = new Ear(); </pre>	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; padding: 5px;">Statement</th><th style="text-align: left; padding: 5px;">Output</th></tr> </thead> <tbody> <tr> <td style="padding: 5px;">var1.method2();</td><td style="padding: 5px;">Mouth 2/Nose 1</td></tr> <tr> <td style="padding: 5px;">var2.method2();</td><td style="padding: 5px;">Ear 2</td></tr> <tr> <td style="padding: 5px;">var3.method2();</td><td style="padding: 5px;">Mouth 2/Eye 1/Mouth 1</td></tr> <tr> <td style="padding: 5px;">var4.method2();</td><td style="padding: 5px;">Compiler Error</td></tr> <tr> <td style="padding: 5px;">var5.method2();</td><td style="padding: 5px;">Mouth 2/Nose 1</td></tr> <tr> <td style="padding: 5px;">var6.method2();</td><td style="padding: 5px;">Ear 2</td></tr> <tr> <td style="padding: 5px;">var1.method3();</td><td style="padding: 5px;">Compiler Error</td></tr> </tbody> </table>	Statement	Output	var1.method2();	Mouth 2/Nose 1	var2.method2();	Ear 2	var3.method2();	Mouth 2/Eye 1/Mouth 1	var4.method2();	Compiler Error	var5.method2();	Mouth 2/Nose 1	var6.method2();	Ear 2	var1.method3();	Compiler Error
Statement	Output																
var1.method2();	Mouth 2/Nose 1																
var2.method2();	Ear 2																
var3.method2();	Mouth 2/Eye 1/Mouth 1																
var4.method2();	Compiler Error																
var5.method2();	Mouth 2/Nose 1																
var6.method2();	Ear 2																
var1.method3();	Compiler Error																

Grading: 1 point for each correct answer. Partial match gives 1 full point.

CSC-133 (Spring 2019)
Attendance Quiz # 4

Student Name: _____

Inheritance and Polymorphism (20 points). Consider the following classes. In the table below, indicate in the right-hand column the output produced by the statement in the left-hand column. If the statement produces more than one line of output, indicate the line breaks with slashes as in "a / b / c" to indicate three lines of output with "a" followed by "b" followed by "c". If the statement causes an error, fill in the right-hand column with the phrase "error" to indicate this.

“S.o.pln” means “System.out.println”.

<pre> public class Eye extends Mouth { public void method1() { S.o.pln("Eye 1"); super.method1(); } } public class Mouth { public void method1() { S.o.pln("Mouth 1"); } public void method2() { S.o.pln("Mouth 2"); method1(); } } public class Nose extends Eye { public void method1() { S.o.pln("Nose 1"); } public void method3() { S.o.pln("Nose 3"); } } public class Ear extends Eye { public void method2() { S.o.pln("Ear 2"); } public void method3() { S.o.pln("Ear 3"); } } </pre>	<p>The following variables are defined:</p> <p>Mouth var1 = new Nose(); Ear var2 = new Ear(); Mouth var3 = new Eye(); Object var4 = new Mouth(); Eye var5 = new Nose(); Mouth var6 = new Ear();</p> <table border="0" style="width: 100%;"> <thead> <tr> <th style="text-align: left; width: 40%;"><u>Statement</u></th><th style="text-align: left; width: 60%;"><u>Output</u></th></tr> </thead> <tbody> <tr> <td>var1.method1();</td><td>_____</td></tr> <tr> <td>var2.method1();</td><td>_____</td></tr> <tr> <td>var3.method1();</td><td>_____</td></tr> <tr> <td>var1.method2();</td><td>_____</td></tr> <tr> <td>var2.method2();</td><td>_____</td></tr> <tr> <td>var3.method2();</td><td>_____</td></tr> <tr> <td>var4.method2();</td><td>_____</td></tr> <tr> <td>var5.method2();</td><td>_____</td></tr> <tr> <td>var6.method2();</td><td>_____</td></tr> <tr> <td>var1.method3();</td><td>_____</td></tr> </tbody> </table>	<u>Statement</u>	<u>Output</u>	var1.method1();	_____	var2.method1();	_____	var3.method1();	_____	var1.method2();	_____	var2.method2();	_____	var3.method2();	_____	var4.method2();	_____	var5.method2();	_____	var6.method2();	_____	var1.method3();	_____
<u>Statement</u>	<u>Output</u>																						
var1.method1();	_____																						
var2.method1();	_____																						
var3.method1();	_____																						
var1.method2();	_____																						
var2.method2();	_____																						
var3.method2();	_____																						
var4.method2();	_____																						
var5.method2();	_____																						
var6.method2();	_____																						
var1.method3();	_____																						

Setsuna Chiu

CSC 133

Spring 2019

Attendance Quiz #5

Starter.java

```
package com.mycompany.pointer;

import static com.codename1.ui.CN.*;
import com.codename1.ui.Form;
import com.codename1.ui.Dialog;
import com.codename1.ui.plaf.UIManager;
import com.codename1.ui.util.Resources;
import com.codename1.io.Log;
import com.codename1.ui.Toolbar;

public class Starter {

    private Form current;
    private Resources theme;

    public void init(Object context) {
        theme = UIManager.initFirstTheme("/theme");

        // Enable Toolbar on all Forms by default
        Toolbar.setGlobalToolbar(true);

        // Pro only feature, uncomment if you have a pro subscription
        Log.bindCrashProtection(true);
    }

    public void start() {
        if(current != null){
            current.show();
            return;
        }
        new PointerListenerFormWithOverride();
    }

    public void stop() {
        current = getCurrentForm();
        if(current instanceof Dialog) {
            ((Dialog)current).dispose();
            current = getCurrentForm();
        }
    }

    public void destroy() {
    }
}
```

PointerListenerFormWithOverride.java

```
package com.mycompany.pointer;

import com.codename1.charts.util.ColorUtil;
import com.codename1.ui.Component;
import com.codename1.ui.Container;
import com.codename1.ui.Form;
import com.codename1.ui.events.ActionEvent;
import com.codename1.ui.events.ActionListener;
import com.codename1.ui.layouts.BorderLayout;
import com.codename1.ui.plaf.Border;

public class PointerListenerFormWithOverride extends Form{
    public PointerListenerFormWithOverride() {
        setLayout(new BorderLayout());

        Container pressContainer = new Container();
        Container dragContainer = new Container();
        Container releaseContainer = new Container();

        pressContainer.addPointerPressedListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent evt) {
                System.out.println("Pointer in (WEST REGION) PRESSED x
and y: " + evt.getX() + " " + evt.getY());
            }
        });

        dragContainer.addPointerDraggedListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent evt) {
                System.out.println("Pointer in (CENTER REGION) DRAGGED x
and y: " + evt.getX() + " " + evt.getY());
            }
        });

        releaseContainer.addPointerReleasedListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent evt) {
                System.out.println("Pointer in (EAST REGION) RELEASED x
and y : " + evt.getX() + " " + evt.getY());
            }
        });

        pressContainer.getAllStyles().setPadding(Component.LEFT, 300);
        pressContainer.getAllStyles().setBorder(Border.createLineBorder(2,
ColorUtil.BLUE));

        releaseContainer.getAllStyles().setPadding(Component.RIGHT, 300);
        releaseContainer.getAllStyles().setBorder(Border.createLineBorder(2,
ColorUtil.GREEN));

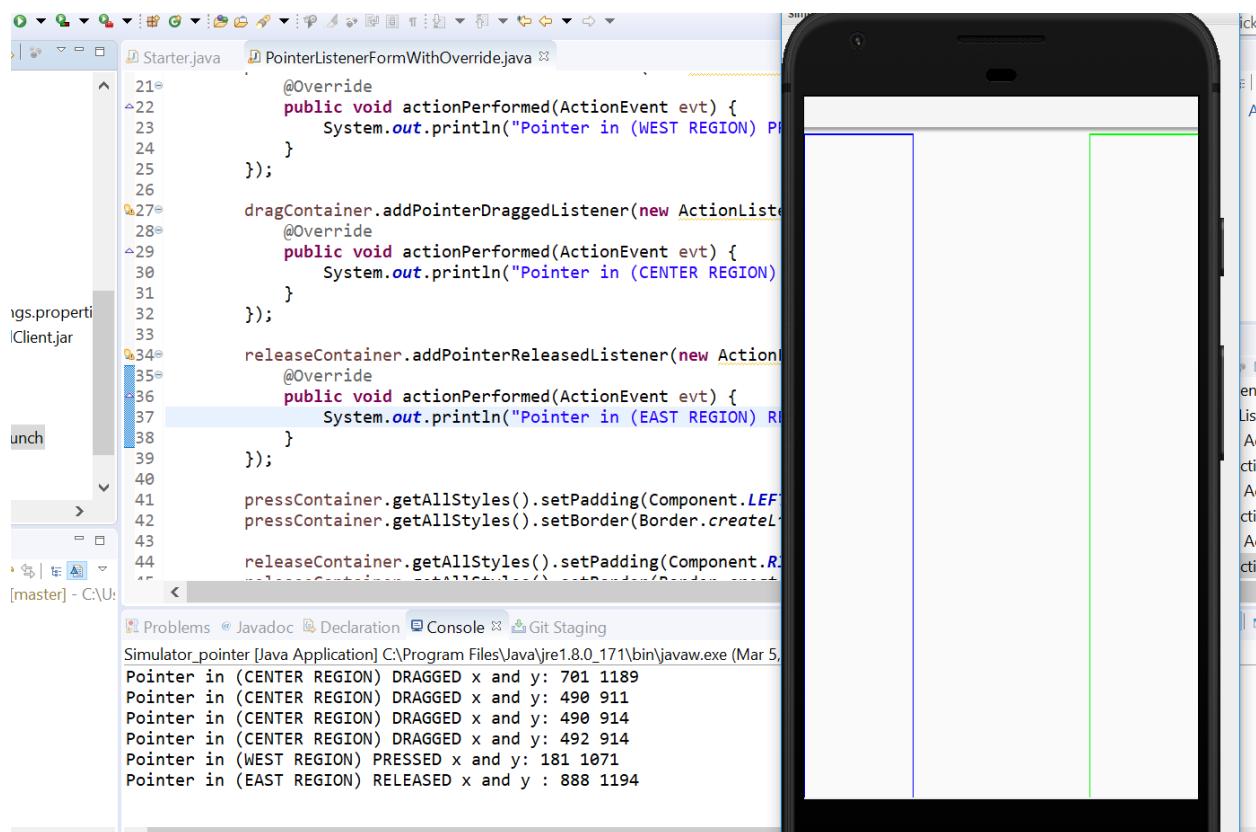
        this.add(BorderLayout.WEST, pressContainer);
```

```
        this.add(BorderLayout.CENTER, dragContainer);
        this.add(BorderLayout.EAST, releaseContainer);

    show();
}

}
```

Output



PointerListenerForm Class:

```
package com.mycompany.eventprog;

import com.codename1.ui.Container;
import com.codename1.ui.Form;
import com.codename1.ui.Label;
import com.codename1.ui.events.ActionEvent;
import com.codename1.ui.events.ActionListener;
import com.codename1.ui.layouts.BorderLayout;
import com.codename1.charts.util.ColorUtil;

public class PointerListenerForm extends Form {
    public PointerListenerForm() {
        setLayout(new BorderLayout());

        Container pressableContainer = new Container();
        pressableContainer.getAllStyles().setBgColor(ColorUtil.rgb(253,231,76));
        pressableContainer.getAllStyles().setBgTransparency(255);

        pressableContainer.getAllStyles().setPaddingRight(this.getContentPane().getWidth()/3);
        PointerPressedListener myPressedListener = new PointerPressedListener();
        pressableContainer.addPointerPressedListener(myPressedListener);
        this.add(BorderLayout.WEST, pressableContainer);

        Container draggableContainer = new Container();
        draggableContainer.setLayout(new BorderLayout());
        draggableContainer.getAllStyles().setBgColor(ColorUtil.rgb(155,197,61));
        draggableContainer.getAllStyles().setBgTransparency(255);

        draggableContainer.getAllStyles().setPaddingRight(this.getContentPane().getWidth()/3);
        PointerDraggedListener myDraggedListener = new PointerDraggedListener();
        draggableContainer.addPointerDraggedListener(myDraggedListener);
        this.add(BorderLayout.CENTER, draggableContainer);

        Container releaseableContainer = new Container();
        releaseableContainer.setLayout(new BorderLayout());

        releaseableContainer.getAllStyles().setBgColor(ColorUtil.rgb(229,89,52));
        releaseableContainer.getAllStyles().setBgTransparency(255);

        releaseableContainer.getAllStyles().setPaddingRight(this.getContentPane().getWidth()/3);
        PointerReleasedListener myReleasedListener = new
        PointerReleasedListener();
        releaseableContainer.addPointerReleasedListener(myReleasedListener);
```

```
        this.add(BorderLayout.EAST, releaseableContainer);

        show();
    }

}

class PointerPressedListener implements ActionListener {
    public void actionPerformed(ActionEvent evt) {
        System.out.println("Pointer Pressed: " + evt.getX() + " " + evt.getY());
    }
}

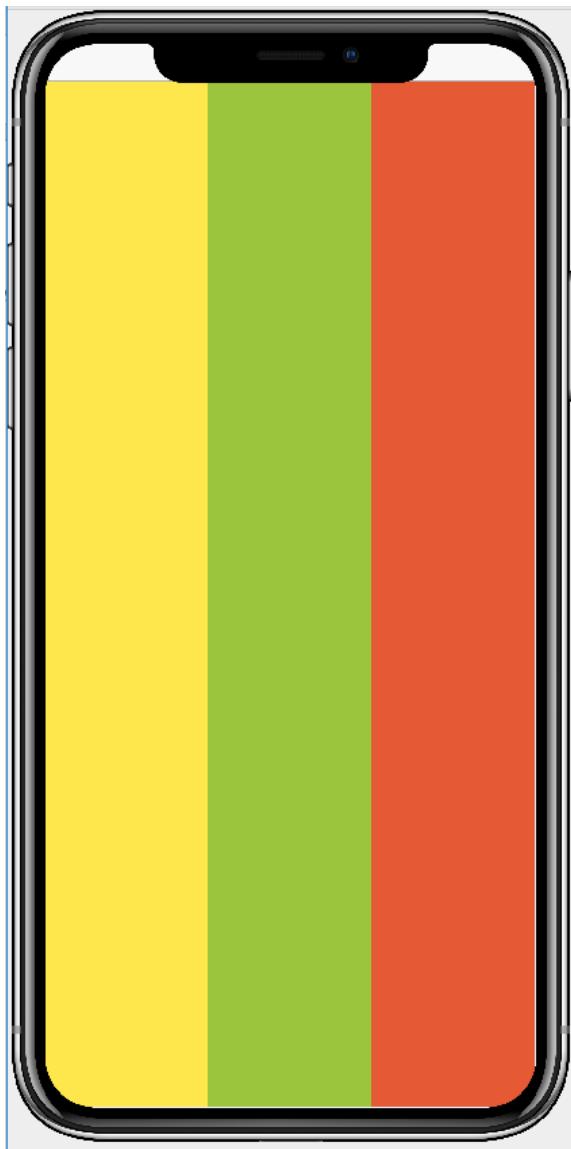
class PointerDraggedListener implements ActionListener {
    public void actionPerformed(ActionEvent evt) {
        System.out.println("Pointer Dragged: " + evt.getX() + " " + evt.getY());
    }
}

class PointerReleasedListener implements ActionListener {
    public void actionPerformed(ActionEvent evt) {
        System.out.println("Pointer Released: " + evt.getX() + " " +
evt.getY());
    }
}
```

Sample Outputs:

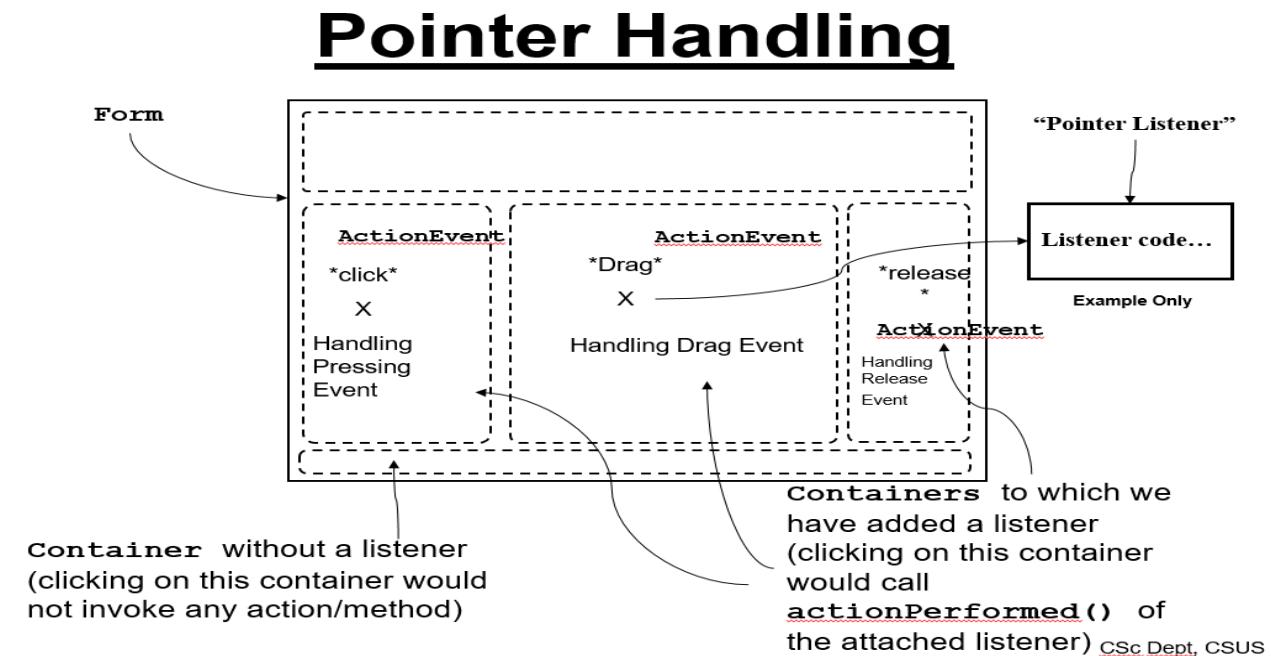
```
Simulator_EventProj [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (Mar 5, 2019, 9:05:17 PM)
```

```
Pointer Pressed: 212 1602
Pointer Pressed: 210 1530
Pointer Pressed: 210 1530
Pointer Released: 1067 1385
Pointer Released: 1067 1385
Pointer Released: 1067 1385
Pointer Dragged: 594 1123
Pointer Dragged: 528 1431
Pointer Dragged: 533 1396
Pointer Dragged: 538 1365
Pointer Dragged: 538 1350
```



Please write a CN1 program using the description below:

Describe below is a structure of a program which implements a Graphical User Interface (GUI) consisting of a form with the following layout and event handling functionalities.



The handling for the events are satisfied by the System.out.println text displaying of which event takes place “Pressed from the EAST region”, “Dragged from Center region”, “Released from the West region”. You are required to output ONLY one action event per region.

Here is an example of an output: (DEMO ONLY - DO NOT USE THIS FOR YOUR WORK)



Sidlak Malaki

CSC 133 – 04

Professor Doan Nguyen

3 May 2019

Attendance Quiz #10

Window(left, right, bottom, top) = (4, 8, 2, 6)

Line, L1, with points P1(9, 1), P2(3, 5)

C1(P1) = 0011

C2(P2) = 0001

0011 & 1000 = 0000

P1 below window:

$$y = 2$$

$$x = 9 + (2 - 1) / ((1 - 5) / (9 - 3)) = 7.5$$

$$P1(7.5, 2)$$

C1(P1) = 0000

C2(P2) = 0001

0000 & 0001 = 0000

C1(P1) == 0 → swap P1 and P2 → P1(3, 5), P2(7.5, 2)

P1 left of window:

$$x = 4$$

$$y = 5 + ((2 - 5) / (7.5 - 3))(4 - 3) = 4.3$$

$$P1(4, 4.33)$$

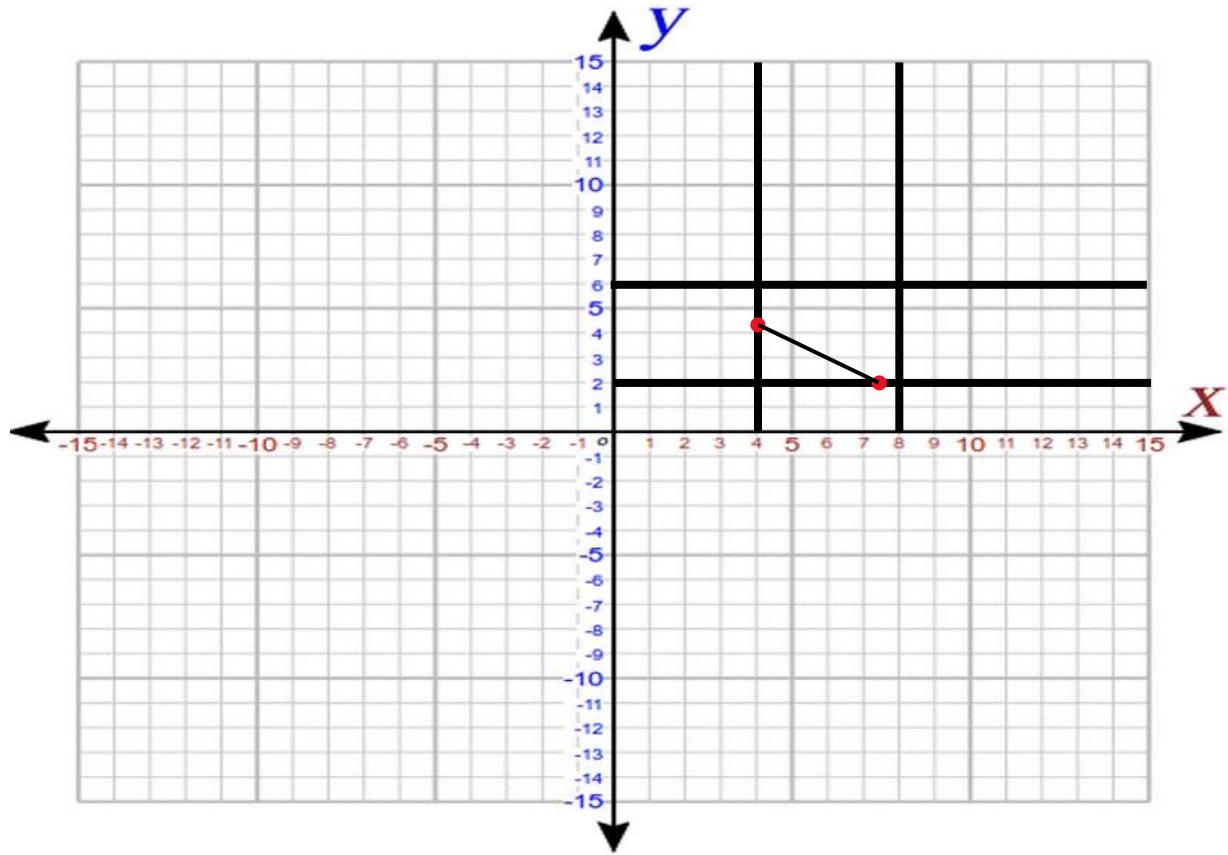
C1(P1) = 0000

C2(P2) = 0000

Both codes = 0 → trivial acceptance

Final Result:

L1 with points P1(4, 4.3) and P2(7.5, 2)



Dennis Dang
CSC 133 – 04
Dr. Doan Nguyen

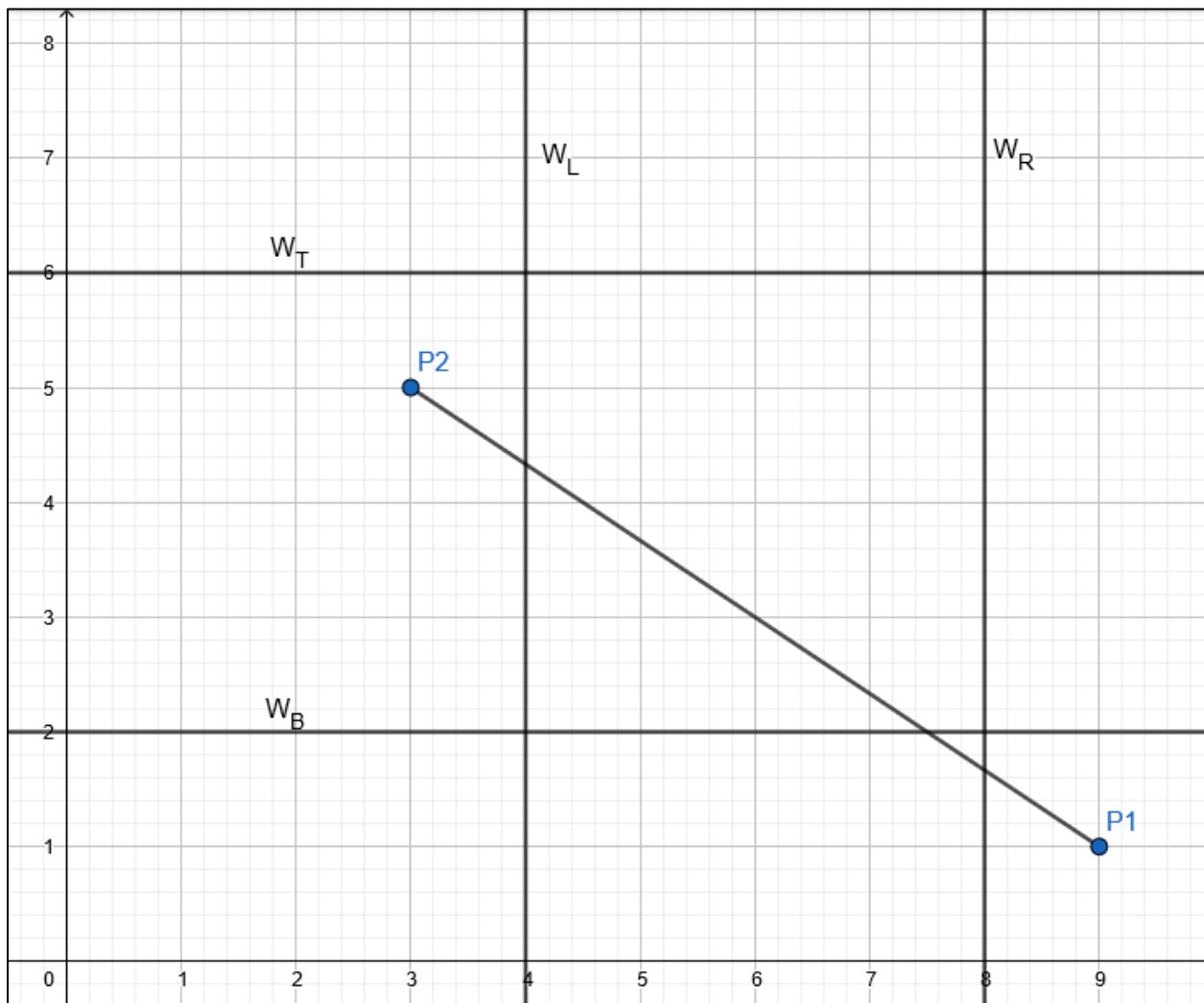
Attendance Quiz #10

Given:

$$\text{Window}(\text{Left}, \text{Right}, \text{Bottom}, \text{Top}) = (4, 8, 2, 6)$$

P1: (9,1)

P2: (3,5)



Here are the steps using the Cohen-Sutherland Clipping Algorithm:

Loop 1:

1. Initialize c1 and c2.
 $c1 = 0\ 0\ 1\ 1, c2 = 1\ 0\ 0\ 0$
2. Check if $(c1 \&& c2) != 0$
From this step, the result is 0 0 0 0. Therefore, do not return null.
3. c1 is **not** 0 0 0 0.
4. Intersect p1 with window

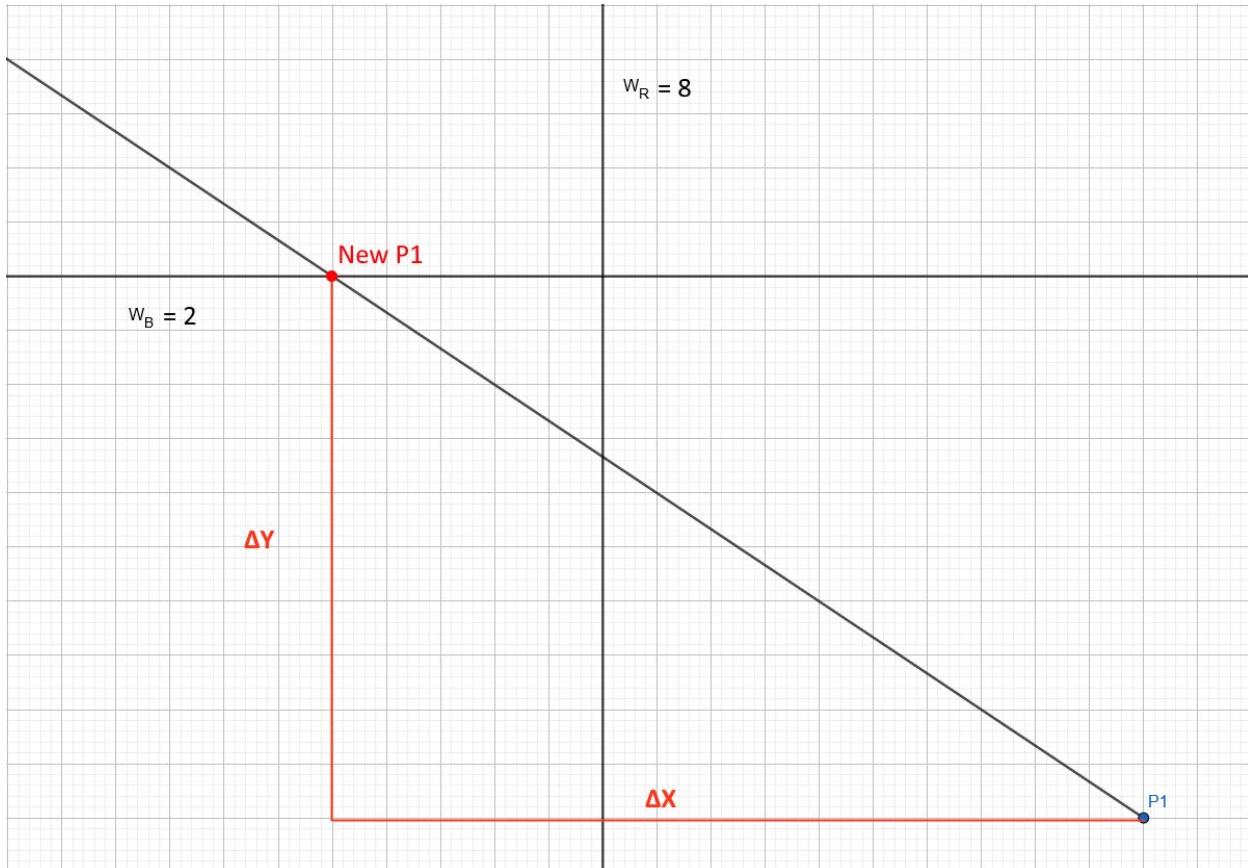
$$\text{Slope} = \frac{5-1}{3-9} = \frac{4}{-6} = -\frac{2}{3} \quad \Delta x = \frac{\Delta y}{\text{Slope}} \quad P_{\text{newY}} = 2$$

$$\Delta y = P1_x - W_R = 1 \rightarrow \Delta x = \frac{1}{-\frac{2}{3}} = \frac{-3}{2}$$

$$P_{\text{newX}} = P1_x + \Delta x = 9 - \frac{3}{2} = 7.5$$

P1: (7.5, 2)

5. $c1 = 0\ 0\ 0\ 0$



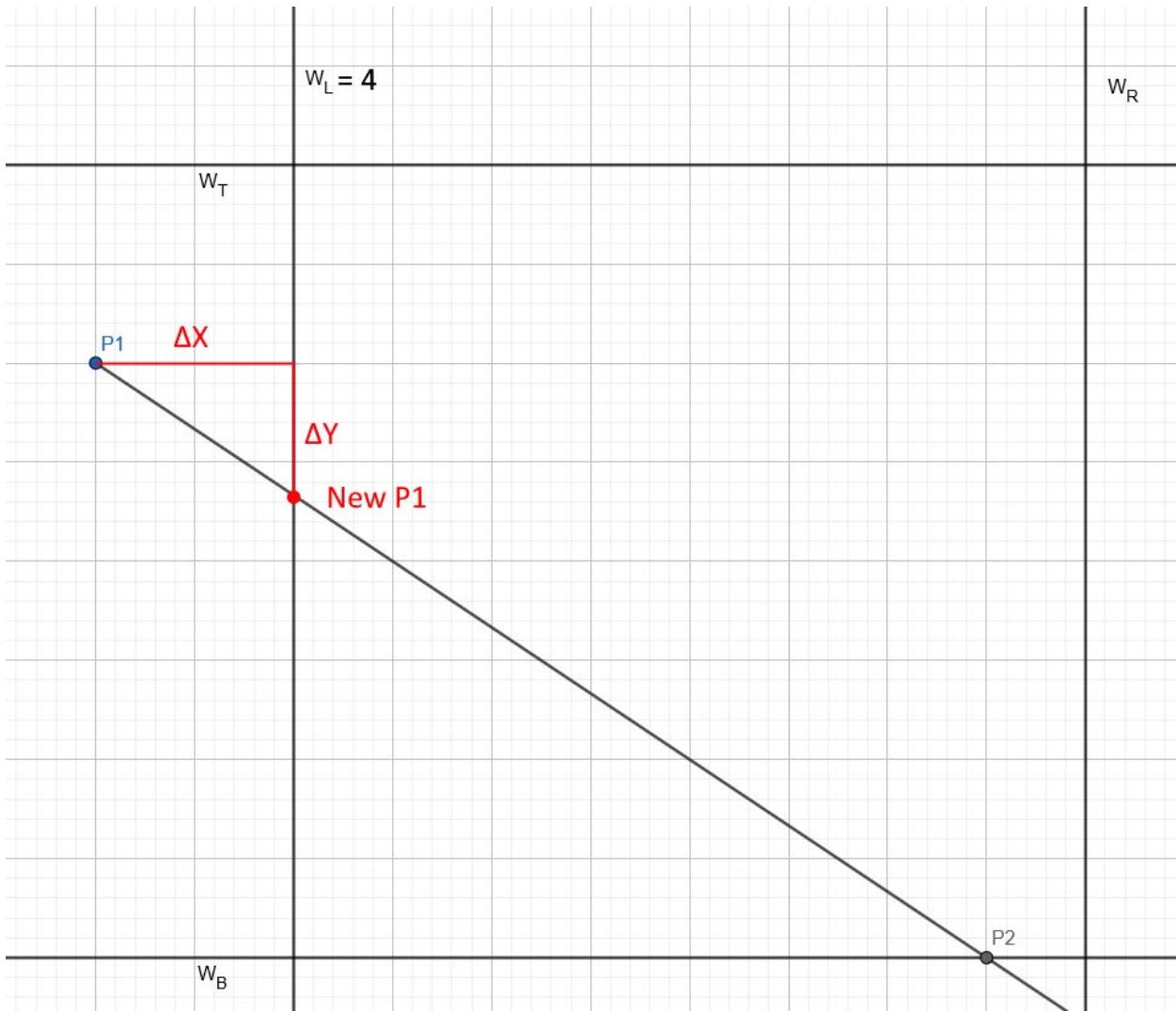
Loop 2

1. Check if $(c1 \&& c2) != 0$
From this step, the result is 0 0 0 0. Therefore, do not return null.
2. $c1$ is 0 0 0 0. So, swap P1 with P2.

P1: (3,5)

P2: (7.5, 2)

Here is an updated image with the swap.



3. Intersect p1 with window

$$\text{Slope} = \frac{2-5}{7.5-3} = \frac{-3}{4.5} \quad P_{\text{new}X} = 4$$

$$\Delta x = W_L - P1_x = 4 - 3 = 1$$

$$\text{Since } \Delta y = \Delta x * \text{Slope}, \quad \Delta y = 1 * \frac{-3}{4.5}$$

$$P_{newY} = P1_y + \Delta y = 5 - \frac{3}{4.5} = 4. \overline{33}$$

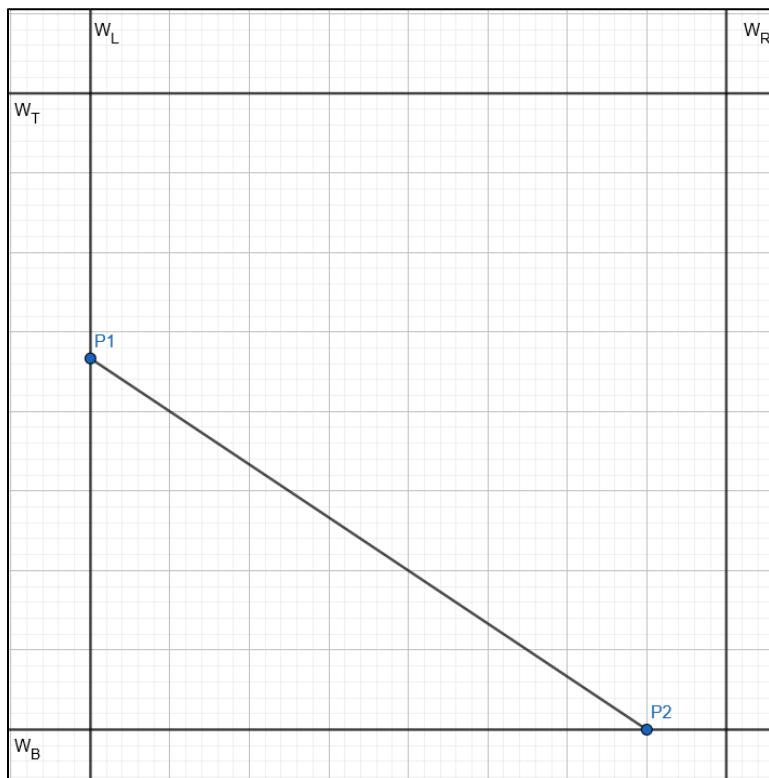
P1: (4, 4. $\overline{33}$)

4. c1 = 0 0 0 0

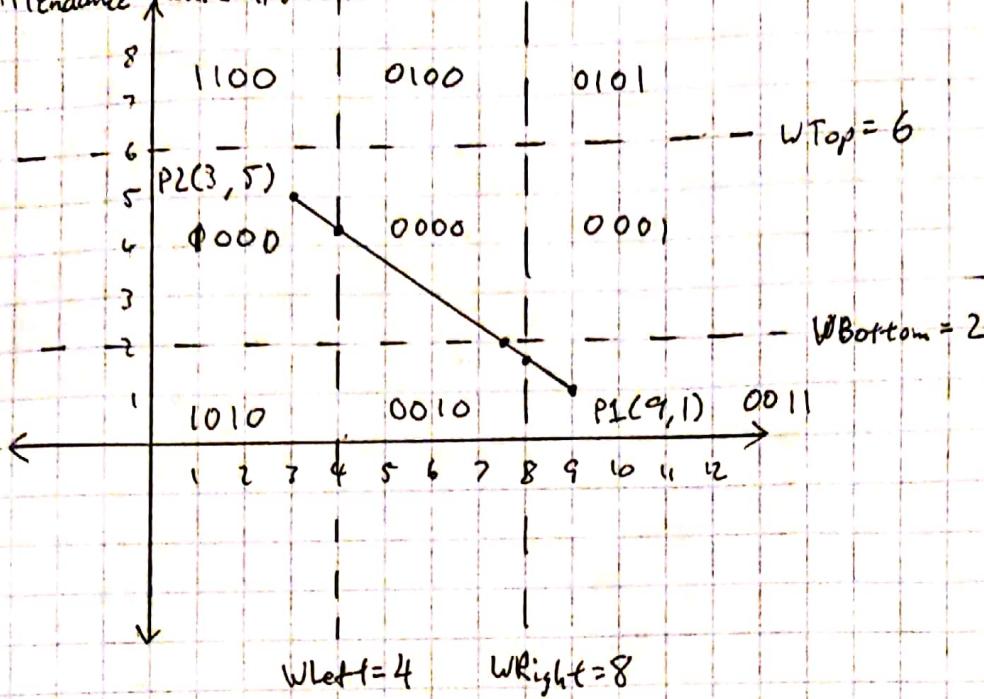
Loop 3: NOT(c1&c2) evaluates to false (c1 = 0000, c2 = 0000). Exit the loop.

We can then draw the clipped line from (4, 4. $\overline{33}$) to (7.5, 2).

Clipped Line:



Shanchern Alan Lin
Attendance Quiz #10



First Clip

$$c_1 = 0011$$

$$c_2 = \emptyset 000$$

$$p_1 = (8, 1.667)$$

$$p_2 = (3, 5)$$

Second Clip

$$c_1 = 0010$$

$$c_2 = \emptyset 000$$

$$p_1 = (7.5, 2)$$

$$p_2 = (3, 5)$$

Third Clip

$$c_1 = 0000$$

$$c_2 = 1000$$

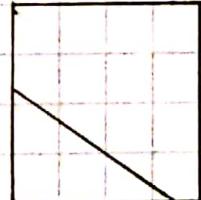
Swap

$$c_1 = 1000$$

$$c_2 = 0000$$

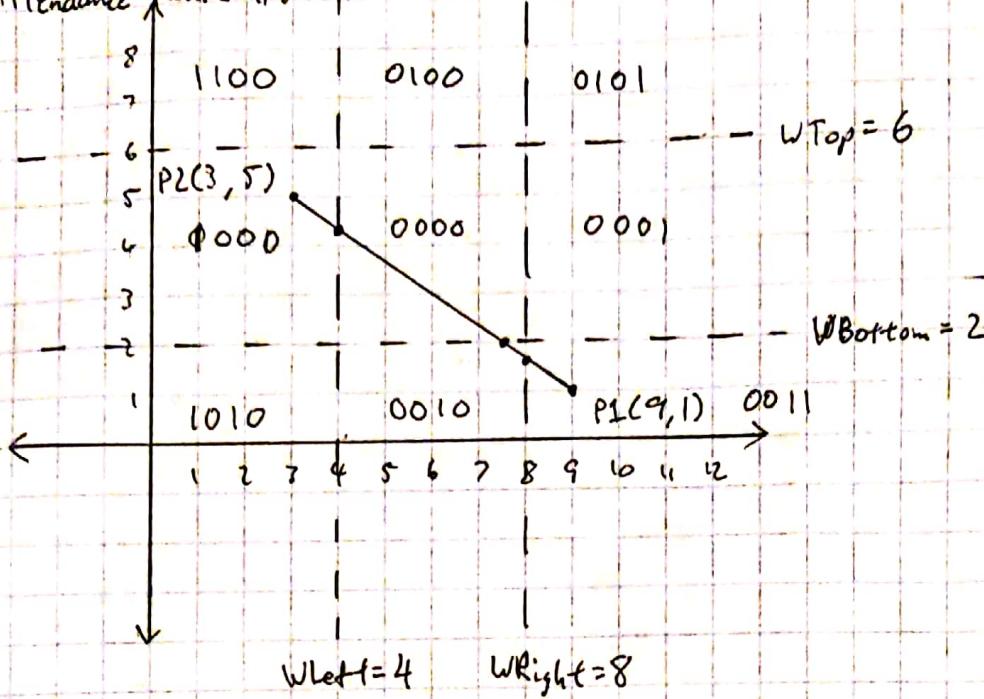
$$\begin{array}{l} p_1 = (4, 4.33) \\ p_2 = (7.5, 2) \end{array}$$

new Line



new Line $L(4, 4.33), (7.5, 2)$

Shanchern Alan Lin
Attendance Quiz #10



First Clip

$$c_1 = 0011$$

$$c_2 = \emptyset 000$$

$$p_1 = (8, 1.667)$$

$$p_2 = (3, 5)$$

Second Clip

$$c_1 = 0010$$

$$c_2 = \emptyset 000$$

$$p_1 = (7.5, 2)$$

$$p_2 = (3, 5)$$

Third Clip

$$c_1 = 0000$$

$$c_2 = 1000$$

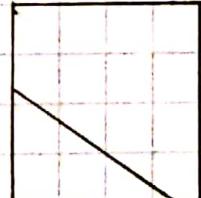
Swap

$$c_1 = 1000$$

$$c_2 = 0000$$

$$\begin{array}{l} p_1 = (4, 4.33) \\ p_2 = (7.5, 2) \end{array}$$

new Line



new Line $L(4, 4.33), (7.5, 2)$

Overview

- CSC 133 Midterm Result
- Module 10 – Introduction to Interactive Techniques

Midterm Result

Statistics (100 Points Total)

Section 2

Maximum Value	95/100
Minimum Value	29/100
Average	59.68
# Submission	40

Section 4

Maximum Value	84/100
Minimum Value	0/100
Average	53.0
# Submission	37

Redo Opportunity (One time)

- Optional (not required)
- Redo multiple true/false/short answer questions (Problem 1), and problems 2,3,4.
- Provide written **TYPED** answers (2-3 pages in PDF format). Also attach your program(s) and its output (for Problem 2,3,4) and a **scanned original exam (must)**
- Resources: Books, Lecture Notes, debugger, your instructor, tutors.
- 0..10-15 points* (no points to be awarded without seeing a student's progress is being made)



- **Treat this as an individual take home exam (not a group assignment).**

Redo Option Deliverables and Midterm Grade Posting

- Due April 2th (In Canvas) – Absolutely, no late work accepted. A new Canvas column will be created.
 - ✓ Only redo problems you missed!
- Multiple choice questions (Problem 1) required explanation/justification – now that you know the answer!
- Redo Problem 2,3 or 4 is required programming with output result (showing they work!) **with full documentation.** Attached your program and its output along with your PDF document.
- Also, redoing Problem 3 is required drawing the UML diagram using Violet (or Creately). Include the diagram in the PDF document.
- Final Midterm grade will be posted in Canvas approximately **April 9th.**

Problem 1

Sample redo answer: Please be clear and concise

N/A Explain what is Polymorphism in context of CN1/Java? How this concept is being utilized in your assignment 1? Motivate your answer with a short of snippet code.

Polymorphism: In Java/CN1 it means that operation that can be done on various types of objects or an operation that can be done in a variety of ways.

Example: In assignment 1, we have the following: We perform the object moving in a polymorphically-safe way like this. Object is moving can be Asteroid, Missile, and so on.

```
for (int i=0; i<theWorldVector.size(); i++) {
    if (theWorldVector.elementAt(i) instanceof IMovable) {
        IMovable mObj = (IMovable)theWorldVector.elementAt(i);
        mObj.move();
    }
}
```

Grader constructive comment: "...write very vaguely or very little so I had to mark it down since I couldn't be sure that students knew what they were talking about."

Problem 1 (Cont)

Sample redo answer: Please be concise. Show the source of where you redo work from.

2) _A_ What Must a non-abstract child do about an abstract method in its parent class?

Correct answer was A. I answered D initially. I probably went a little too fast on this question because I knew the answer was you have to have a solid defined method when making an abstract subclass concrete. Regardless the keyword override in answer A is what makes the difference. So each subclass must override and implement the abstract class's methods. Information is from the following slide.

4 - Inheritance

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)

35 CSc Dept, CSUS

Problem 1 – A common issue

Provide the expected output for the following program:

```
class Animal {  
    public void move() {  
        System.out.println("Animals can move");  
    }  
}  
  
class Dog extends Animal {  
    public void move() {  
        System.out.println("Dogs can walk and run");  
    }  
    public void bark() {  
        System.out.println("Dogs can bark");  
    }  
}  
  
public class TestDog {  
  
    public static void main(String args[]) {  
        Animal a = new Animal(); // Animal reference and object  
        Animal b = new Dog(); // Animal reference but Dog object  
  
        a.move(); // runs the method in Animal class  
        b.move(); // runs the method in Dog class  
        b.bark();  
    }  
}
```

When the instructor reviewed student exams, many stated outputs for the code.

Output: Syntax error. Cannot find symbol bark. **NO OUTPUT AT ALL.**

Think apparent type vs. actual type!

Problem 2

- Private variables can not gain direct access to from sub class(es). Need to use setter/getter method.

CSC 133 Lecture Notes
3 - OOP Concepts

Access (Visibility) Modifiers

Java:



Modifier	Access Allowed By			
	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<none>	Y	Y*	N	N
private	Y	N	N	N



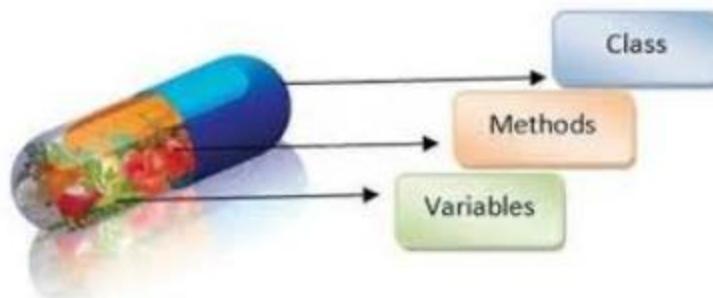
Problem 2 (cont)

- Usage of protected modifier to fix the issue?

If there's going to be a public getter anyway, why would you want to expose the field itself more widely than absolutely necessary? That means it's immediately writable by subclasses.

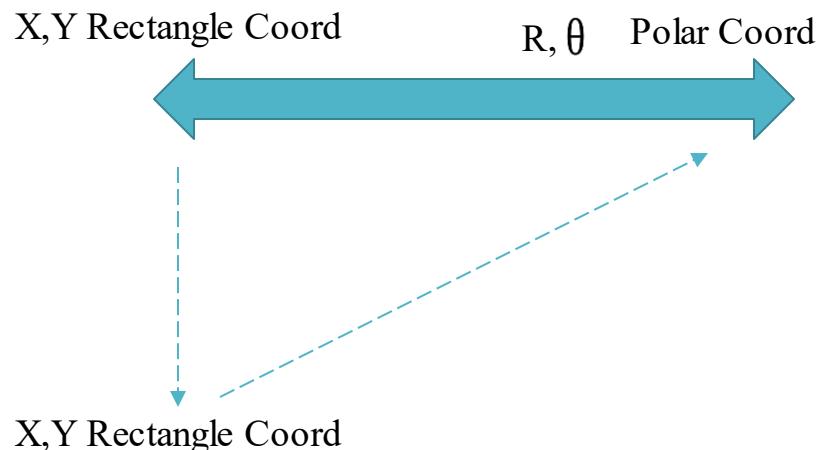
<https://stackoverflow.com/questions/2279662/java-protected-fields-vs-public-getters>

- Broke encapsulation rule (data and methods collocated)



Problem 2 (cont)

- Synchronization issue between Polar and Rectangular Coordinates.
- Without having the subclass overriding parent setX/setY and to update the polar coordinate, we would have the following scenario:



Problem 3 – Design Pattern & UML Class Diagram

- Grader Constructive Comment: For the third question, there were a lot of points missed to the (1) relation directions going the wrong way and for either (2) missing the interfaces, (3) implementing them incorrectly, (4) incorrectly usage of aggregation/composition.
- Instructor comment: Many students suggested Iterative Design Pattern (Behavior). But the problem mentioned creational one!
- Singleton Design Pattern (Creational). But there is no trait on creating one SINGLE object.
- But, we covered one more creational design pattern in class? Please review lecture note again.

Problem 3 – UML Class Diagram

- Please use proper relations. See module 3 (slide 22-32).
- Please be informed interface(s) relation correctly specified in UML diagram (see module 4).
- Identify of all the key components including multiplicity and arrows direction.
- Redraw the work using Violet or Creately.

Problem 4 – Command Design Pattern

- Grader constructive comment: I would advise the students to carefully read the question to understand what it is asking. Many students had unnecessary code written. The notes had some examples of similar questions that are a good reference.

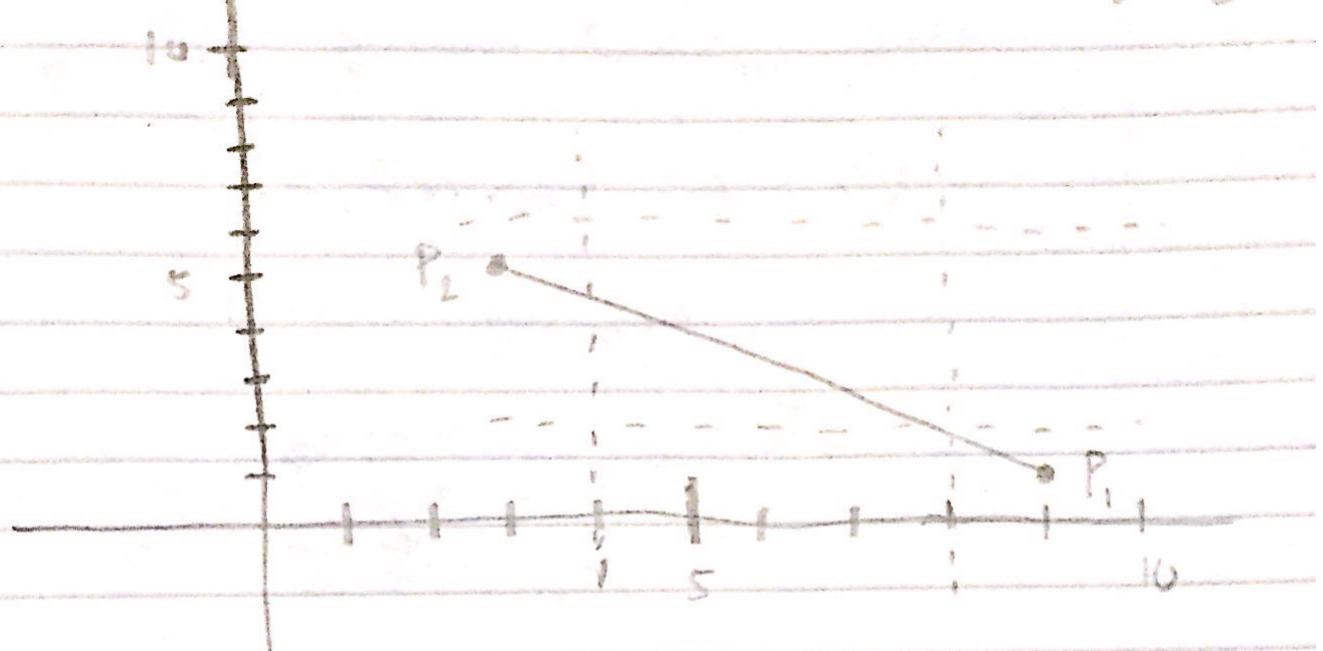
Problem 4 – Command Design Pattern

- The first step is to have an understanding of what is Command Design Pattern.
- Need to know the steps of how to create container.
- Knowledge about layout: Box Layout and Flow Layout.
- Know how add the container to the form, creating buttons , etc
- Know to assigning keys to the button, and add commands to the button.

Please review lectures on Command Design Pattern on: Design Pattern Part II (Week 7), slide 10-11 (Cut/Delete example) and Attendance Quiz 5 (take home) - before midterm.

Final Exam Schedule

Class	Class Title	Exam Date	Exam Time	Exam Room
<u>CSC</u> <u>133-02</u> <u>(36458)</u>	Obj-Oriented Cmptr Graph (Discussion)	5/14/2019, Tuesday	10:15AM - 12:15PM	Riverside Hall 5029
<u>CSC</u> <u>133-04</u> <u>(32708)</u>	Obj-Oriented Cmptr Graph (Discussion)	5/16/2019, Thursday	12:45PM - 2:45PM	Riverside Hall 1002



$$1) C_1 = 0011 \text{ (P}_1\text{)}$$

$$C_2 = 1000 \text{ (P}_2\text{)}$$

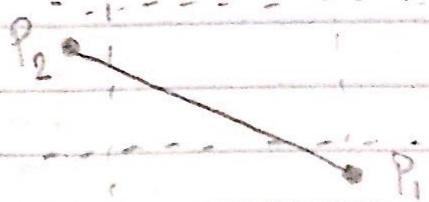
$$C_1 \& C_2 = 0000$$

line crosses W right

$$m = \frac{1-5}{9-3} = -4/6$$

$$x = 8$$

$$y = 1 - \frac{4}{6}(8-9) = \frac{5}{3}$$



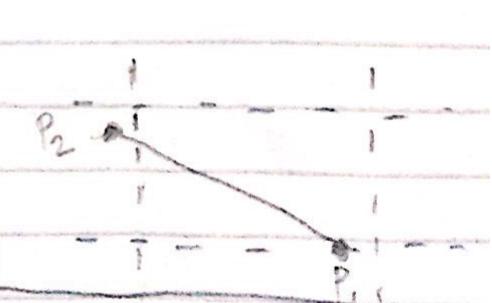
$$2) C_1 = 0010$$

$$C_2 = 1000$$

$$C_1 \& C_2 = 0000$$

crosses W bottom

$$m = \frac{\frac{5}{3}-5}{8-3} = \frac{-10/3}{5} = -\frac{2}{3}$$



$$y = 2$$

$$x = 8 + \left(2 - \frac{5}{3}\right) / -\frac{2}{3} = 8 - \frac{1}{2} = 7.5$$

$$3) C_1 = 0000 \quad C_1 \& C_2 = 0000$$

$$C_2 = 1000$$

since $C_1 = 0$, swap P_1 and P_2

$$P_1(3, 5) \quad P_2(7.5, 2)$$

crosses WLeft

$$x = 4$$

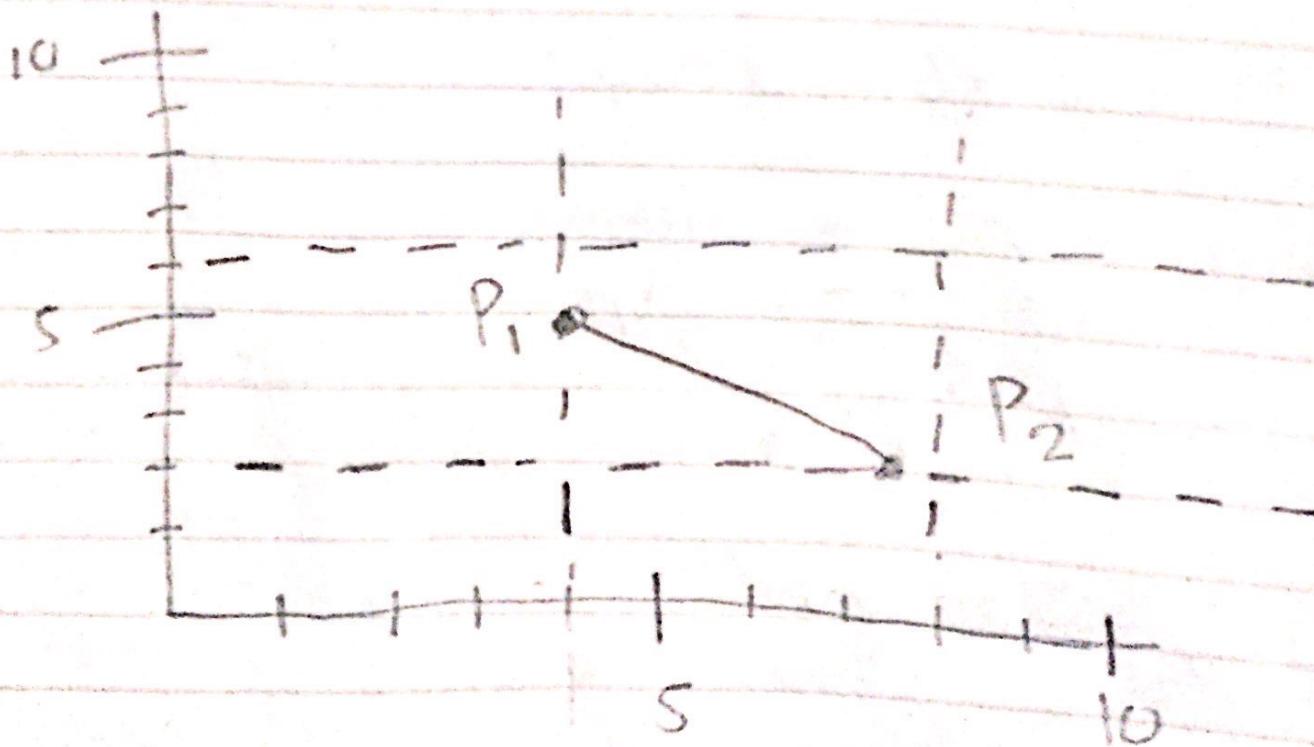
$$y = 5 - \frac{2}{3}(4 - 3) \approx 4.3$$

$$m = \frac{5 - 15}{3 - 7.5} = \frac{-10}{-4.5} = -\frac{2}{3}$$

$$L_1 = 0000$$

$$L_1 \& L_2 = 0000$$

$$L_2 = 0000$$



$$P_1(4, 4.3) \quad P_2(7, 2)$$

$$P_2 = (9, 1) \quad C_1 = 0011 \quad m = -\frac{2}{3}$$

$$P_2 = (3, 5) \quad C_2 = 1000$$

$$y = 2 \quad x = 9 + (2-1)/-\frac{2}{3} = 9 - \frac{3}{2} = 7\frac{1}{2}$$

$$P_1 = (7\frac{1}{2}, 2) \quad C_1 = 00000$$

Final

$$\text{swap} \quad P_1 = (3, 5) \quad C_1 = 1000$$

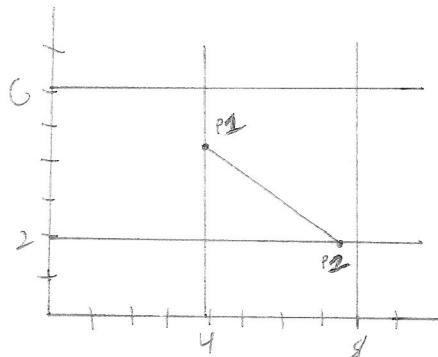
$$P_1 = (4, 4\frac{1}{3}) \quad C_1 = 00000$$

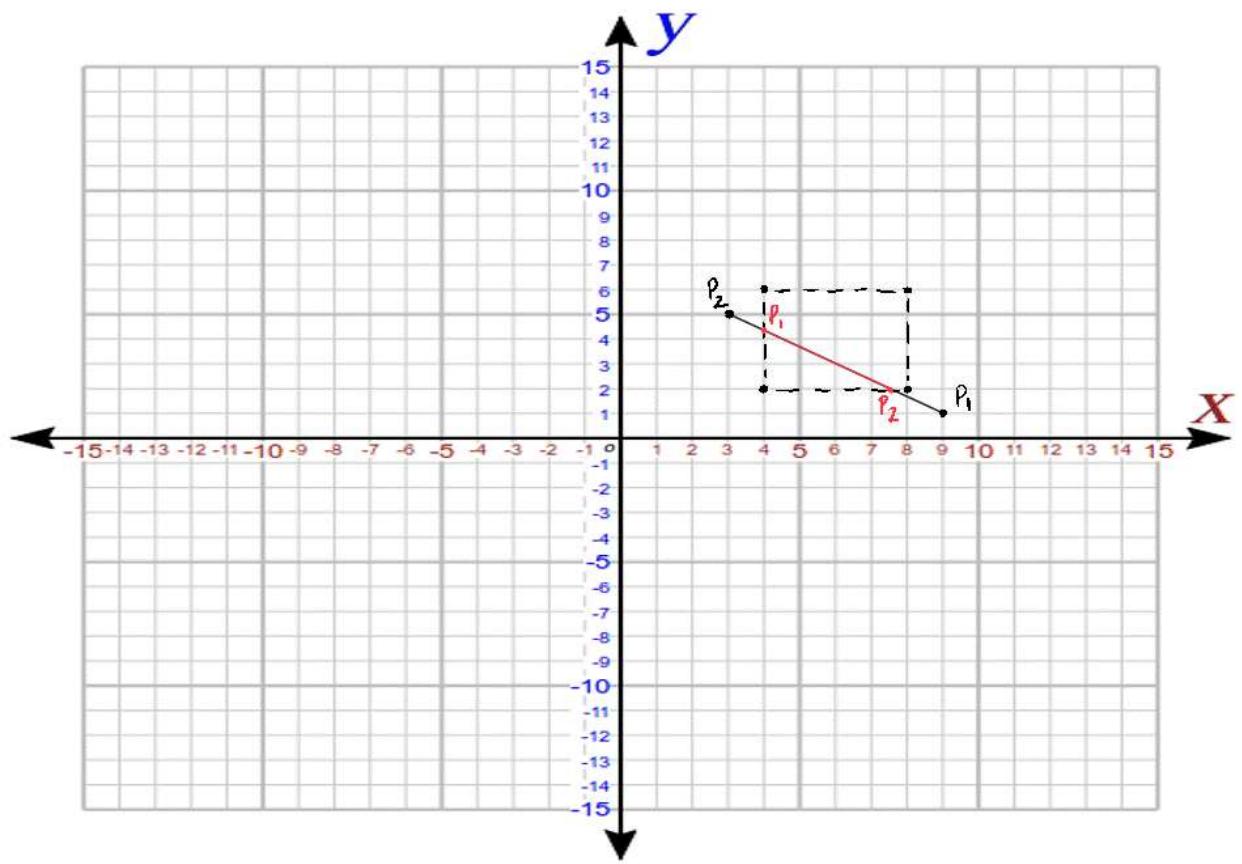
$$P_2 = (7\frac{1}{2}, 2) \quad C_2 = 00000$$

$$P_2 = (7\frac{1}{2}, 2) \quad C_2 = 00000$$

$$x = 4 \quad y = 5 + (-\frac{2}{3})(4-3) = 4\frac{1}{3}$$

$$P_1 = (4, 4\frac{1}{3}) \quad C_1 = 00000$$





$$P_1(9, 1) \quad P_2(3, 5) \quad m = (P_2y - P_1y) / (P_2x - P_1x)$$
$$= (5 - 1) / (3 - 9)$$

$$y = 2$$
$$= 4 / -6$$
$$= 2 / -3$$
$$x = 9 + (2 - 1) / (2 / -3)$$
$$= 9 + (1) / (2 / -3)$$
$$= 9 + (-1.5)$$
$$= 7.5$$

$$P_1(7.5, 2)$$

$(c1 == 0)$, so the points swap!

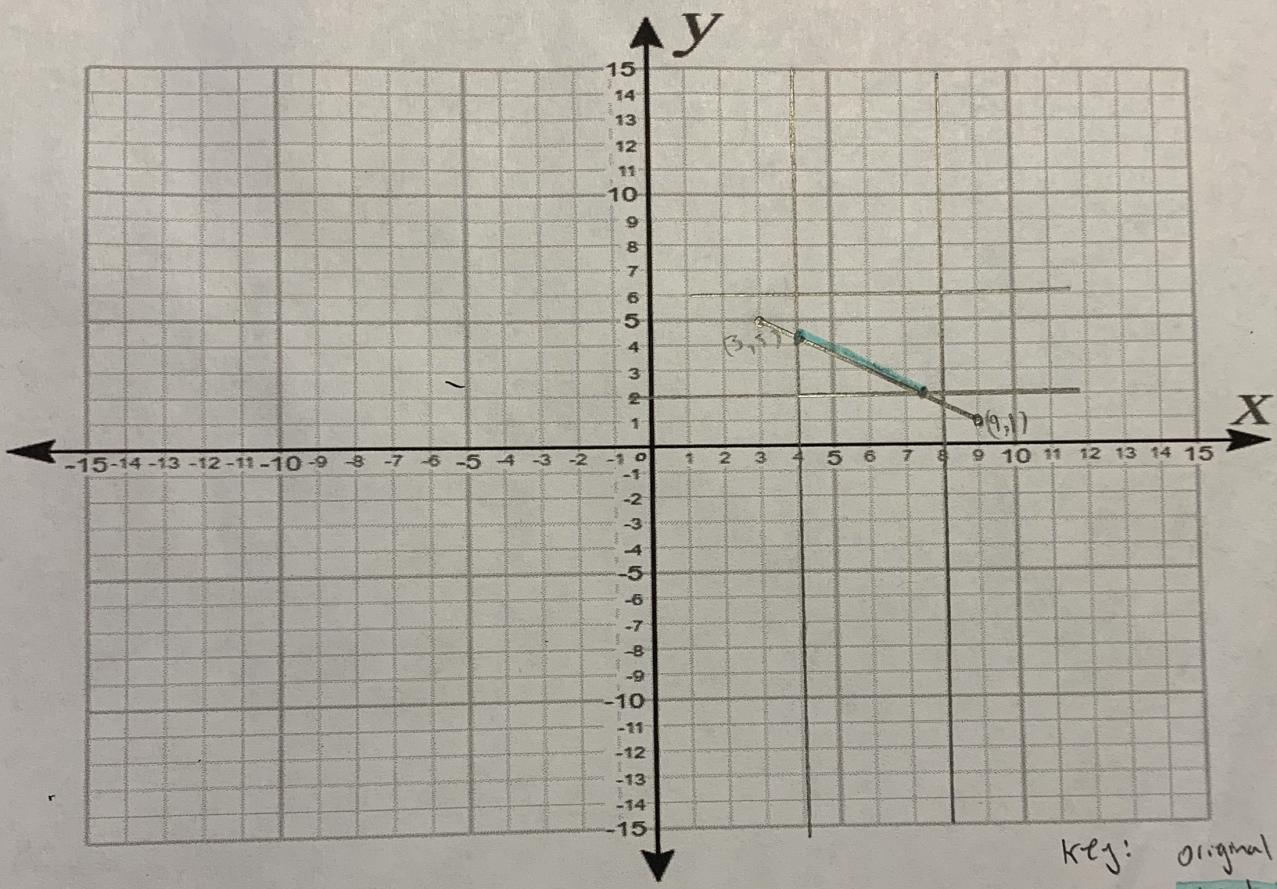
$$P_1(3, 5) \quad P_2(7.5, 2)$$
$$m = (2 - 5) / (7.5 - 3)$$

$$x = 4 \quad = -3 / 4.5$$

$$y = 5 + (-3 / 4.5)(4 - 3)$$
$$= 5 + (-3 / 4.5)(1)$$
$$= 5 + (-3 / 4.5)$$
$$= 4.\overline{33}$$

$$\boxed{P_1(4, 4.\overline{33}) \quad P_2(7.5, 2)}$$

Scott N.
O'Hearn



Key: original line
clipped line

$$P_1(9, 1) = 0011$$

$$P_2(3, 5) = 1000$$

↓ intersect with window

$$y = y_1 + m(x - x_1) = 1 + (-2/3)(8 - 9) = 1.33$$

$$x = x_1 + (y - y_1)/m = 9 + (2 - 1.33)/(-2/3)$$

1st
clip

$$P_1(8, 1.33)$$

$$P_2(3, 5)$$

↓ intersect with window
swap

$$y = y_1 + m(x - x_1) = 5 + (-2/3)(8 - 3) = 4.33$$

$$x = x_1 + (y - y_1)/m = 8 + (2 - 4.33)/(-2/3) = 7.5$$

2nd
clip

$$P_1(3, 5)$$

↓ intersect with window

$$x = 4$$

$$y = y_1 + m(x - x_1) = 5 + (-2/3)(4 - 3) = 4.67$$

$(1, 4.67) + (7.5, 2)$

3rd
clip

$$P_1(4, 4.67)$$

$$P_2(2, 7.5)$$

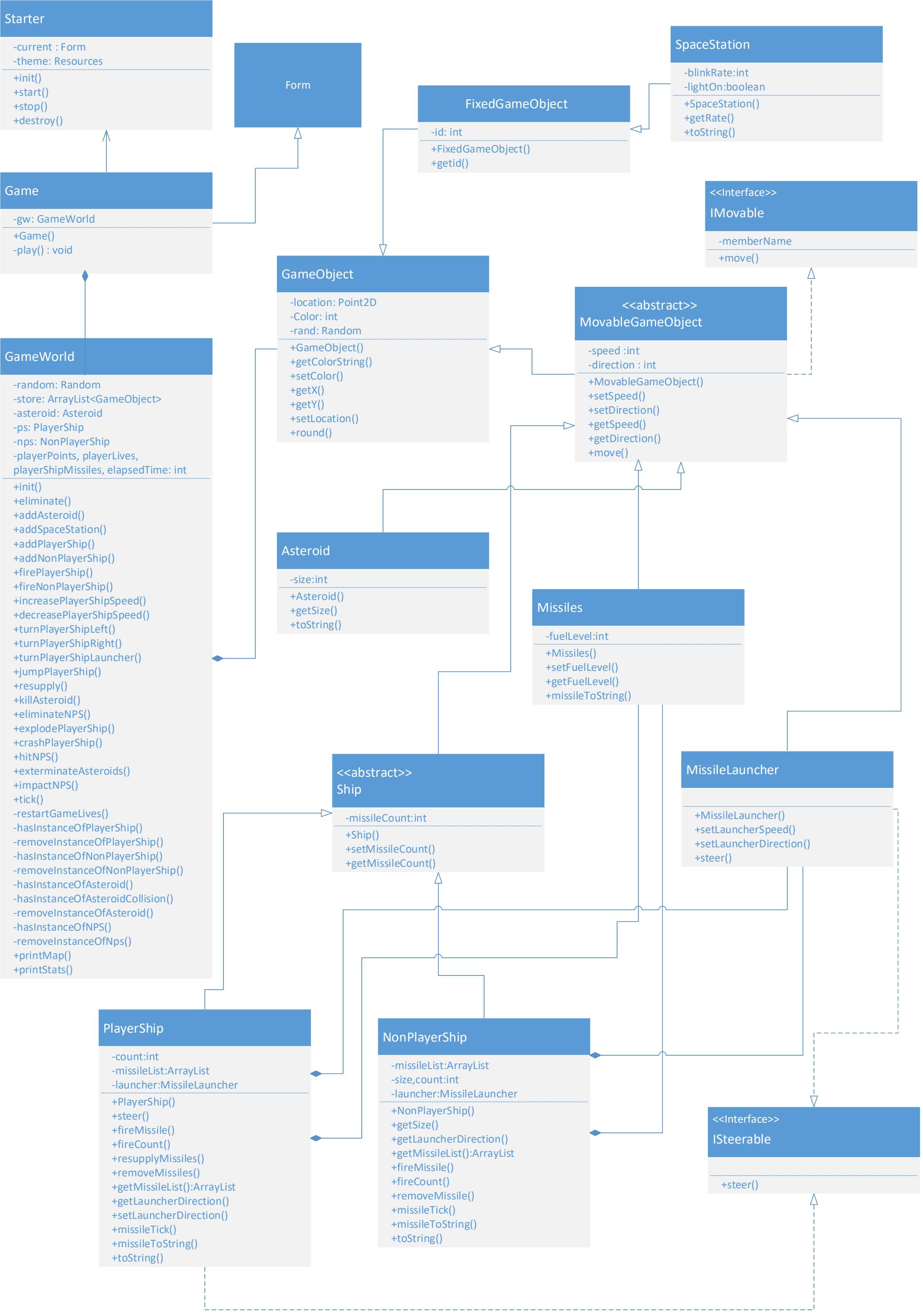
3 clips total

Row-major order

$$[x \ y \ z] * \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad \begin{aligned} x' &= x * a + y * d + z * g \\ y' &= x * b + y * e + z * h \\ z' &= x * c + y * f + z * i \end{aligned}$$

Column-major order

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \begin{aligned} x' &= a * x + b * y + c * z \\ y' &= d * x + e * y + f * z \\ z' &= g * x + h * y + i * z \end{aligned}$$



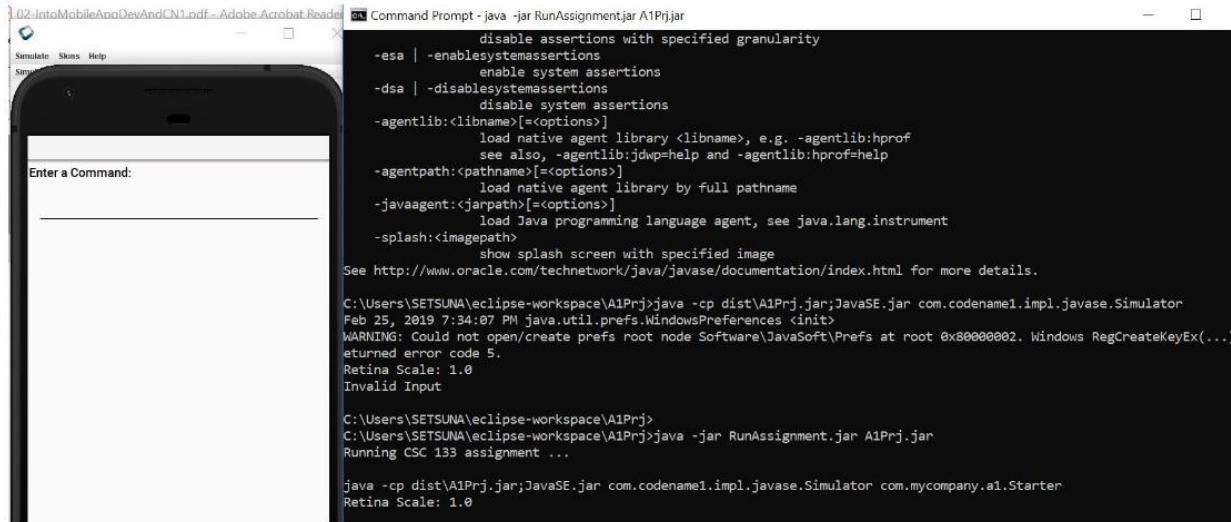
Setsuna Chiu

CSC133

Spring 2019

Assignment 1 Running Test Cases

Test Case 1) Launch the program with java -jar RunAssignment.jar A1Prj.jar



The screenshot shows a mobile application interface with a command line window. The command line displays the Java documentation for assertions and then the execution of the program. The output shows the creation of objects like an asteroid, player ship, and non-player ship, along with missile launches and map details.

```
1.02-IntroMobileAppDevAndCN1.pdf - Adobe Acrobat Reader
Simulate Skins Help
Simulator
Enter a Command:
C:\ Command Prompt - java -jar RunAssignment.jar A1Prj.jar
  disable assertions with specified granularity
  -esa | -enablesystemassertions
  enable system assertions
  -dsa | -disablesystemassertions
  disable system assertions
  -agentlib:<library>[=<options>]
    load native agent library <libraryname>, e.g. -agentlib:hprof
    see also, -agentlib:jdwtp=help and -agentlib:hprof=help
  -agentpath:<pathname>[=<options>]
    load native agent library by full pathname
  -javaagent:<jarpath>[=<options>]
    load Java programming language agent, see java.lang.instrument
  -splash:<imagepath>
    show splash screen with specified image
See http://www.oracle.com/technetwork/java/javase/documentation/index.html for more details.

C:\Users\SETSUNA\eclipse-workspace\A1Prj>java -cp dist\A1Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator
Feb 25, 2019 7:34:07 PM java.util.prefs.WindowsPreferences <init>
WARNING: Could not open/create prefs root node Software\JavaSoft\Prefs at root 0x80000002. Windows RegCreateKeyEx(... returned error code 5.
Retina Scale: 1.0
Invalid Input

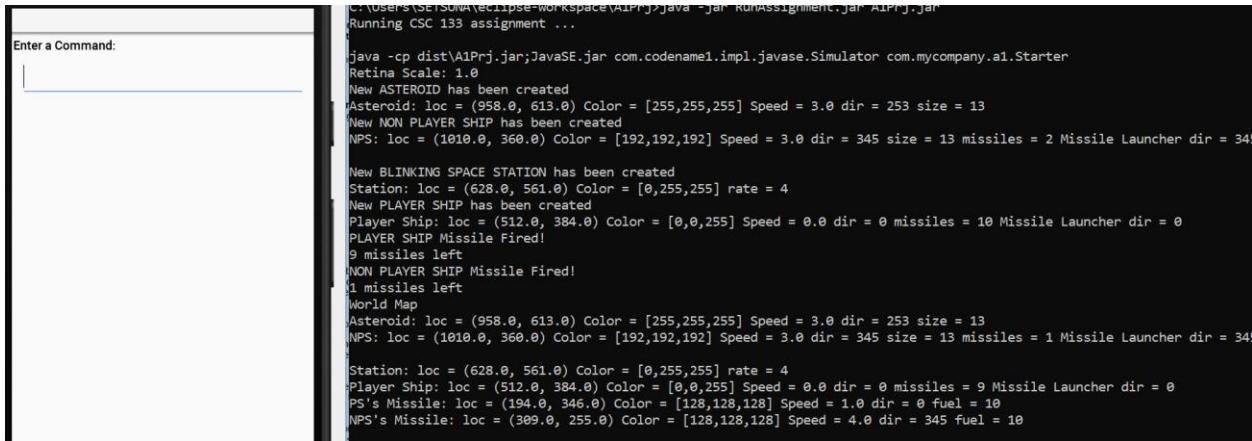
C:\Users\SETSUNA\eclipse-workspace\A1Prj>
C:\Users\SETSUNA\eclipse-workspace\A1Prj>java -jar RunAssignment.jar A1Prj.jar
Running CSC 133 assignment ...

java -cp dist\A1Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a1.Starter
Retina Scale: 1.0
New ASTEROID has been created
Asteroid: loc = (958.0, 613.0) Color = [255,255,255] Speed = 3.0 dir = 253 size = 13
New NON PLAYER SHIP has been created
NPS: loc = (1010.0, 360.0) Color = [192,192,192] Speed = 3.0 dir = 345 size = 13 missiles = 2 Missile Launcher dir = 345

New BLINKING SPACE STATION has been created
Station: loc = (628.0, 561.0) Color = [0,255,255] rate = 4
New PLAYER SHIP has been created
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 10 Missile Launcher dir = 0
PLAYER SHIP Missile Fired!
9 missiles left
NON PLAYER SHIP Missile Fired!
1 missiles left
World Map
Asteroid: loc = (958.0, 613.0) Color = [255,255,255] Speed = 3.0 dir = 253 size = 13
NPS: loc = (1010.0, 360.0) Color = [192,192,192] Speed = 3.0 dir = 345 size = 13 missiles = 1 Missile Launcher dir = 345

Station: loc = (628.0, 561.0) Color = [0,255,255] rate = 4
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 9 Missile Launcher dir = 0
PS's Missile: loc = (194.0, 346.0) Color = [128,128,128] Speed = 1.0 dir = 0 fuel = 10
NPS's Missile: loc = (309.0, 255.0) Color = [128,128,128] Speed = 4.0 dir = 345 fuel = 10
```

Test Case 2) Introduce All Objects to the game



The screenshot shows a mobile application interface with a command line window. The command line displays the Java documentation and then the execution of the program. The output shows the creation of various game objects like an asteroid, player ship, non-player ship, station, and player ship, along with their properties and initial states.

```
C:\Users\SETSUNA\eclipse-workspace\A1Prj>java -jar RunAssignment.jar A1Prj.jar
Running CSC 133 assignment ...

java -cp dist\A1Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a1.Starter
Retina Scale: 1.0
New ASTEROID has been created
Asteroid: loc = (958.0, 613.0) Color = [255,255,255] Speed = 3.0 dir = 253 size = 13
New NON PLAYER SHIP has been created
NPS: loc = (1010.0, 360.0) Color = [192,192,192] Speed = 3.0 dir = 345 size = 13 missiles = 2 Missile Launcher dir = 345

New BLINKING SPACE STATION has been created
Station: loc = (628.0, 561.0) Color = [0,255,255] rate = 4
New PLAYER SHIP has been created
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 10 Missile Launcher dir = 0
PLAYER SHIP Missile Fired!
9 missiles left
NON PLAYER SHIP Missile Fired!
1 missiles left
World Map
Asteroid: loc = (958.0, 613.0) Color = [255,255,255] Speed = 3.0 dir = 253 size = 13
NPS: loc = (1010.0, 360.0) Color = [192,192,192] Speed = 3.0 dir = 345 size = 13 missiles = 1 Missile Launcher dir = 345

Station: loc = (628.0, 561.0) Color = [0,255,255] rate = 4
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 9 Missile Launcher dir = 0
PS's Missile: loc = (194.0, 346.0) Color = [128,128,128] Speed = 1.0 dir = 0 fuel = 10
NPS's Missile: loc = (309.0, 255.0) Color = [128,128,128] Speed = 4.0 dir = 345 fuel = 10
```

All game objects introduced to the game – use map command to verify

Test Case 3) Object interaction

'l' command:

```
New PLAYER SHIP has been created
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 1.0 dir = 0 missiles
```

'd' command:

```
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 1.0 dir = 0
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0
```

'l' ell command:

```
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles =
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 359 missiles
```

'>' command:

```
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 359 missiles = 10 Missile Launcher dir = 0
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 359 missiles = 10 Missile Launcher dir = 100
```

'f' command:

```
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 359 missiles = 10 Missiles Fired!
PLAYER SHIP Missile Fired!
9 missiles left
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 359 missiles = 9 Missiles Fired!
PS's Missile: loc = (77.0, 291.0) Color = [128,128,128] Speed = 1.0 dir = 100 fuel = 10
```

'L' command:

```
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 359 missiles = 9 Missile Launcher dir = 100
PS's Missile: loc = (77.0, 291.0) Color = [128,128,128] Speed = 1.0 dir = 100 fuel = 10
New NON PLAYER SHIP has been created
NPS: loc = (486.0, 2.0) Color = [192,192,192] Speed = 7.0 dir = 154 size = 17 missiles = 2 Missile Launcher dir = 154
NON PLAYER SHIP Missile Fired!
1 missiles left
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 359 missiles = 9 Missile Launcher dir = 100
NPS: loc = (486.0, 2.0) Color = [192,192,192] Speed = 7.0 dir = 154 size = 17 missiles = 1 Missile Launcher dir = 154
PS's Missile: loc = (77.0, 291.0) Color = [128,128,128] Speed = 1.0 dir = 100 fuel = 10
NPS's Missile: loc = (772.0, 320.0) Color = [128,128,128] Speed = 8.0 dir = 154 fuel = 10
```

'j' command:

```
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 359 missiles = 9 Missi
NPS: loc = (486.0, 2.0) Color = [192,192,192] Speed = 7.0 dir = 154 size = 17 missiles = 1 M
PS's Missile: loc = (77.0, 291.0) Color = [128,128,128] Speed = 1.0 dir = 100 fuel = 10
NPS's Missile: loc = (772.0, 320.0) Color = [128,128,128] Speed = 8.0 dir = 154 fuel = 10
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 359 missiles = 9 Missi
```

'n' command:

```
PLAYER SHIP Missile Fired!
8 missiles left
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 359 missiles = 8 Missi
NPS: loc = (486.0, 2.0) Color = [192,192,192] Speed = 7.0 dir = 154 size = 17 missiles = 1 M
PS's Missile: loc = (77.0, 291.0) Color = [128,128,128] Speed = 1.0 dir = 100 fuel = 10
PS's Missile: loc = (430.0, 18.0) Color = [128,128,128] Speed = 1.0 dir = 100 fuel = 10
NPS's Missile: loc = (772.0, 320.0) Color = [128,128,128] Speed = 8.0 dir = 154 fuel = 10
Player Ship Missiles Resupplied
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 359 missiles = 10 Miss
NPS: loc = (486.0, 2.0) Color = [192,192,192] Speed = 7.0 dir = 154 size = 17 missiles = 1 M
PS's Missile: loc = (77.0, 291.0) Color = [128,128,128] Speed = 1.0 dir = 100 fuel = 10
PS's Missile: loc = (430.0, 18.0) Color = [128,128,128] Speed = 1.0 dir = 100 fuel = 10
NPS's Missile: loc = (772.0, 320.0) Color = [128,128,128] Speed = 8.0 dir = 154 fuel = 10
```

'k' command:

```
New ASTEROID has been created
Asteroid: loc = (953.0, 84.0) Color = [255,255,255] Speed = 6.0 dir = 313 size = 28
Player Ship Killed Asteroid!
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 359 missiles = 10 Miss
NPS: loc = (486.0, 2.0) Color = [192,192,192] Speed = 7.0 dir = 154 size = 17 missiles = 1 M
PS's Missile: loc = (430.0, 18.0) Color = [128,128,128] Speed = 1.0 dir = 100 fuel = 10
NPS's Missile: loc = (772.0, 320.0) Color = [128,128,128] Speed = 8.0 dir = 154 fuel = 10
```

'e' command:

```
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 359 missiles = 10 Miss
```

'E' command:

```
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 359 missiles = 10 Miss
NPS: loc = (94.0, 499.0) Color = [192,192,192] Speed = 11.0 dir = 128 size = 14 missiles = 1
NPS's Missile: loc = (206.0, 604.0) Color = [128,128,128] Speed = 12.0 dir = 128 fuel = 10
World Map
NPS: loc = (94.0, 499.0) Color = [192,192,192] Speed = 11.0 dir = 128 size = 14 missiles = 1
```

'c' command:

```
New PLAYER SHIP has been created
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 10 Missiles
New ASTEROID has been created
Asteroid: loc = (330.0, 38.0) Color = [255,255,255] Speed = 0.0 dir = 135 size = 8
World Map
NPS: loc = (94.0, 499.0) Color = [192,192,192] Speed = 11.0 dir = 128 size = 14 missiles = 1
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 10 Missiles
Asteroid: loc = (330.0, 38.0) Color = [255,255,255] Speed = 0.0 dir = 135 size = 8
Player Ship Crashed Into Asteroid!
World Map
NPS: loc = (94.0, 499.0) Color = [192,192,192] Speed = 11.0 dir = 128 size = 14 missiles = 1
```

'h' command:

```
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 1
New NON PLAYER SHIP has been created
NPS: loc = (710.0, 429.0) Color = [192,192,192] Speed = 3.0 dir = 312 size = 16 missiles = 1
Player Ship Crashed Into Non Player Ship!
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 1
```

'x' command:

```
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 1
Asteroid: loc = (372.0, 616.0) Color = [255,255,255] Speed = 6.0 dir = 258 size = 13
Asteroid: loc = (919.0, 353.0) Color = [255,255,255] Speed = 1.0 dir = 161 size = 14
Two Asteroids Collided!
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 1
```

'l' aye command:

```
New ASTEROID has been created
Asteroid: loc = (791.0, 144.0) Color = [255,255,255] Speed = 12.0 dir = 339 size = 13
New NON PLAYER SHIP has been created
NPS: loc = (649.0, 626.0) Color = [192,192,192] Speed = 10.0 dir = 67 size = 17 missiles = 1
Asteroid Collided into Non Player Ship!
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 1
```

't' command:

```
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 8 Missile Launcher dir = 0
Asteroid: loc = (108.0, 753.0) Color = [255,255,255] Speed = 7.0 dir = 64 size = 20
Asteroid: loc = (795.0, 347.0) Color = [255,255,255] Speed = 3.0 dir = 270 size = 23
NPS: loc = (63.0, 327.0) Color = [192,192,192] Speed = 0.0 dir = 152 size = 10 missiles = 1 Missile Launcher dir = 152
Station: loc = (599.0, 356.0) Color = [0,255,255] rate = 4
PS's Missile: loc = (852.0, 4.0) Color = [128,128,128] Speed = 1.0 dir = 0 fuel = 10
PS's Missile: loc = (278.0, 246.0) Color = [128,128,128] Speed = 1.0 dir = 0 fuel = 10
NPS's Missile: loc = (458.0, 524.0) Color = [128,128,128] Speed = 1.0 dir = 152 fuel = 10
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 8 Missile Launcher dir = 0
Asteroid: loc = (113.0, 758.0) Color = [255,255,255] Speed = 7.0 dir = 64 size = 20
Asteroid: loc = (793.0, 350.0) Color = [255,255,255] Speed = 3.0 dir = 270 size = 23
NPS: loc = (63.0, 327.0) Color = [192,192,192] Speed = 0.0 dir = 152 size = 10 missiles = 1 Missile Launcher dir = 152
Station: loc = (599.0, 356.0) Color = [0,255,255] rate = 4
PS's Missile: loc = (852.0, 5.0) Color = [128,128,128] Speed = 1.0 dir = 0 fuel = 9
PS's Missile: loc = (278.0, 247.0) Color = [128,128,128] Speed = 1.0 dir = 0 fuel = 9
NPS's Missile: loc = (458.0, 525.0) Color = [128,128,128] Speed = 1.0 dir = 152 fuel = 9
```

'm' command:

```
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 8 Missile Launcher dir = 0
Asteroid: loc = (108.0, 753.0) Color = [255,255,255] Speed = 7.0 dir = 64 size = 20
Asteroid: loc = (795.0, 347.0) Color = [255,255,255] Speed = 3.0 dir = 270 size = 23
NPS: loc = (63.0, 327.0) Color = [192,192,192] Speed = 0.0 dir = 152 size = 10 missiles = 1 Missile Launcher dir = 152
Station: loc = (599.0, 356.0) Color = [0,255,255] rate = 4
PS's Missile: loc = (852.0, 4.0) Color = [128,128,128] Speed = 1.0 dir = 0 fuel = 10
PS's Missile: loc = (278.0, 246.0) Color = [128,128,128] Speed = 1.0 dir = 0 fuel = 10
NPS's Missile: loc = (458.0, 524.0) Color = [128,128,128] Speed = 1.0 dir = 152 fuel = 10
World Map
Player Ship: loc = (512.0, 384.0) Color = [0,0,255] Speed = 0.0 dir = 0 missiles = 8 Missile Launcher dir = 0
Asteroid: loc = (113.0, 758.0) Color = [255,255,255] Speed = 7.0 dir = 64 size = 20
Asteroid: loc = (793.0, 350.0) Color = [255,255,255] Speed = 3.0 dir = 270 size = 23
NPS: loc = (63.0, 327.0) Color = [192,192,192] Speed = 0.0 dir = 152 size = 10 missiles = 1 Missile Launcher dir = 152
Station: loc = (599.0, 356.0) Color = [0,255,255] rate = 4
PS's Missile: loc = (852.0, 5.0) Color = [128,128,128] Speed = 1.0 dir = 0 fuel = 9
PS's Missile: loc = (278.0, 247.0) Color = [128,128,128] Speed = 1.0 dir = 0 fuel = 9
NPS's Missile: loc = (458.0, 525.0) Color = [128,128,128] Speed = 1.0 dir = 152 fuel = 9
```

'p' command:

```
Stats
Player Lives: 2 Player score: 0 Player Missile Count: 8 Time: 1
```

Test Case 4)

'q' command:



(5.6) 3 in the x
-2 in y, then 30° anti-clock about (0,0)

Show all the matrices computation work here: (10 points)

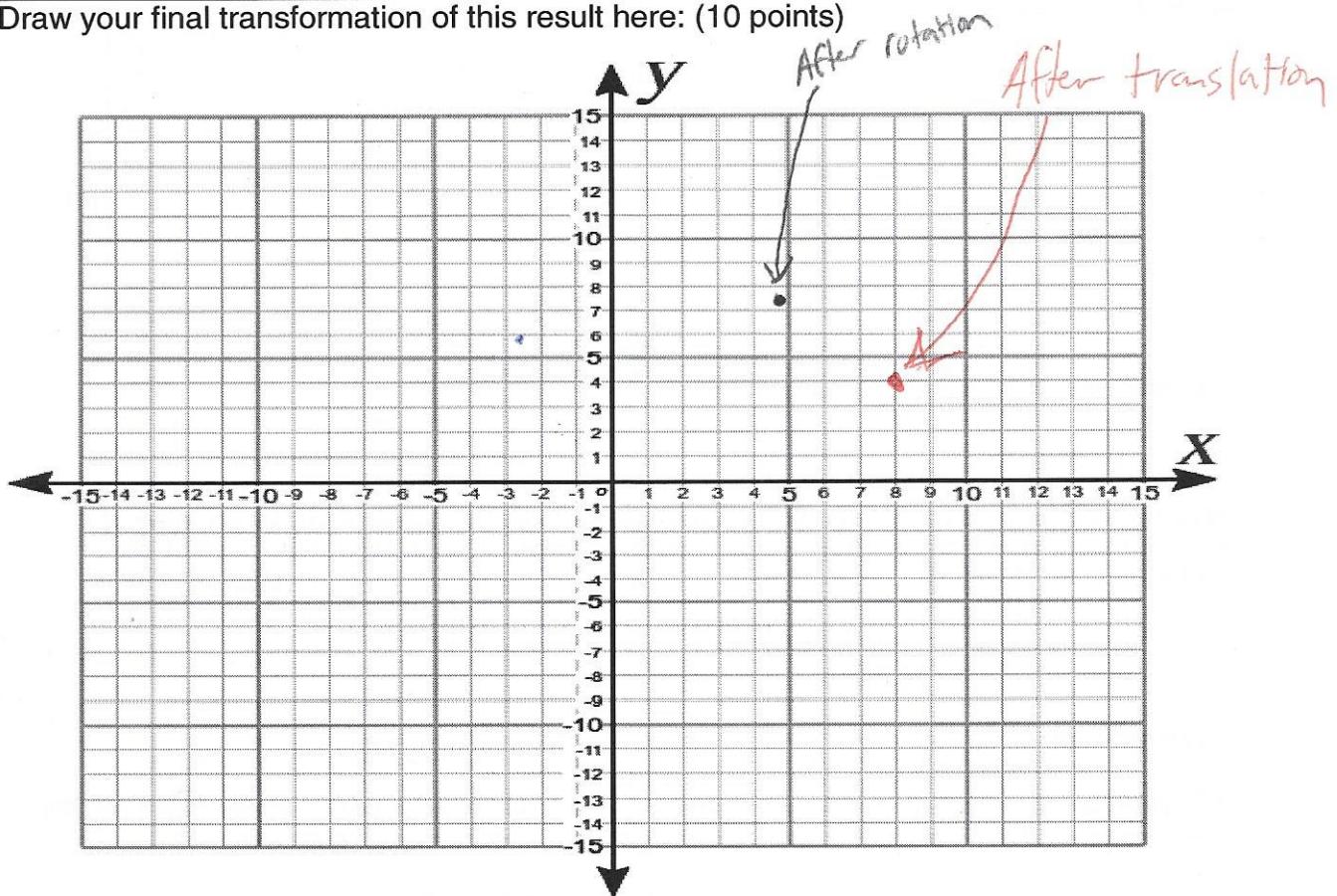
$$T_{(a,b)} = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 5 \\ 6 \\ 1 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 1 \end{pmatrix}$$

\downarrow
 3×3 3×1
 3×1

$$R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$R_{30^\circ} = \begin{pmatrix} \cos(30^\circ) & -\sin(30^\circ) & 0 \\ \sin(30^\circ) & \cos(30^\circ) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 8 \\ 4 \\ 1 \end{pmatrix} = \begin{pmatrix} (8 \cdot \cos(30^\circ)) + (4 \cdot -\sin(30^\circ)) \\ (8 \cdot \sin(30^\circ)) + (4 \cdot \cos(30^\circ)) \\ 1 \end{pmatrix} = \begin{pmatrix} 4.93 \\ 7.46 \\ 1 \end{pmatrix} = (4.93, 7.46)$$

Draw your final transformation of this result here: (10 points)



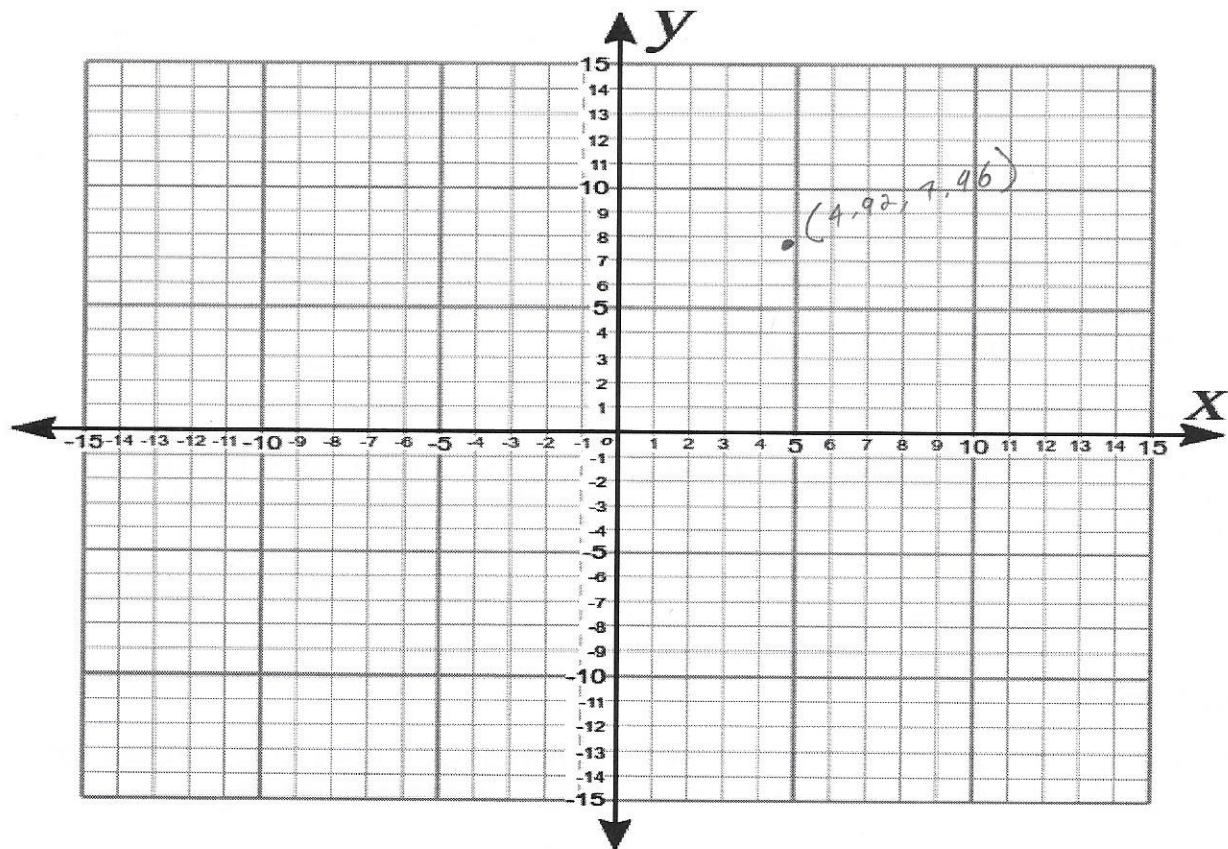
Show all the matrices computation work here: (10 points)

$$\begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 5 \\ 6 \\ 1 \end{bmatrix} = \begin{bmatrix} 5+0+3 \\ 0+6-2 \\ 0+0+1 \end{bmatrix} = \begin{bmatrix} 8 \\ 4 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos(30) & -\sin(30) & 0 \\ \sin(30) & \cos(30) & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 8 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 8\cos(30) - 4\sin(30) \\ 8\sin(30) + 4\cos(30) \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 8\left(\frac{\sqrt{3}}{2}\right) - 4\left(\frac{1}{2}\right) \\ 8\left(\frac{1}{2}\right) + 4\left(\frac{\sqrt{3}}{2}\right) \\ 1 \end{bmatrix} = \begin{bmatrix} 6.92 & -2 \\ 4 & 3.46 \\ 1 \end{bmatrix} = \begin{bmatrix} 4.92 \\ 7.96 \\ 1 \end{bmatrix}$$

Draw your final transformation of this result here: (10 points)

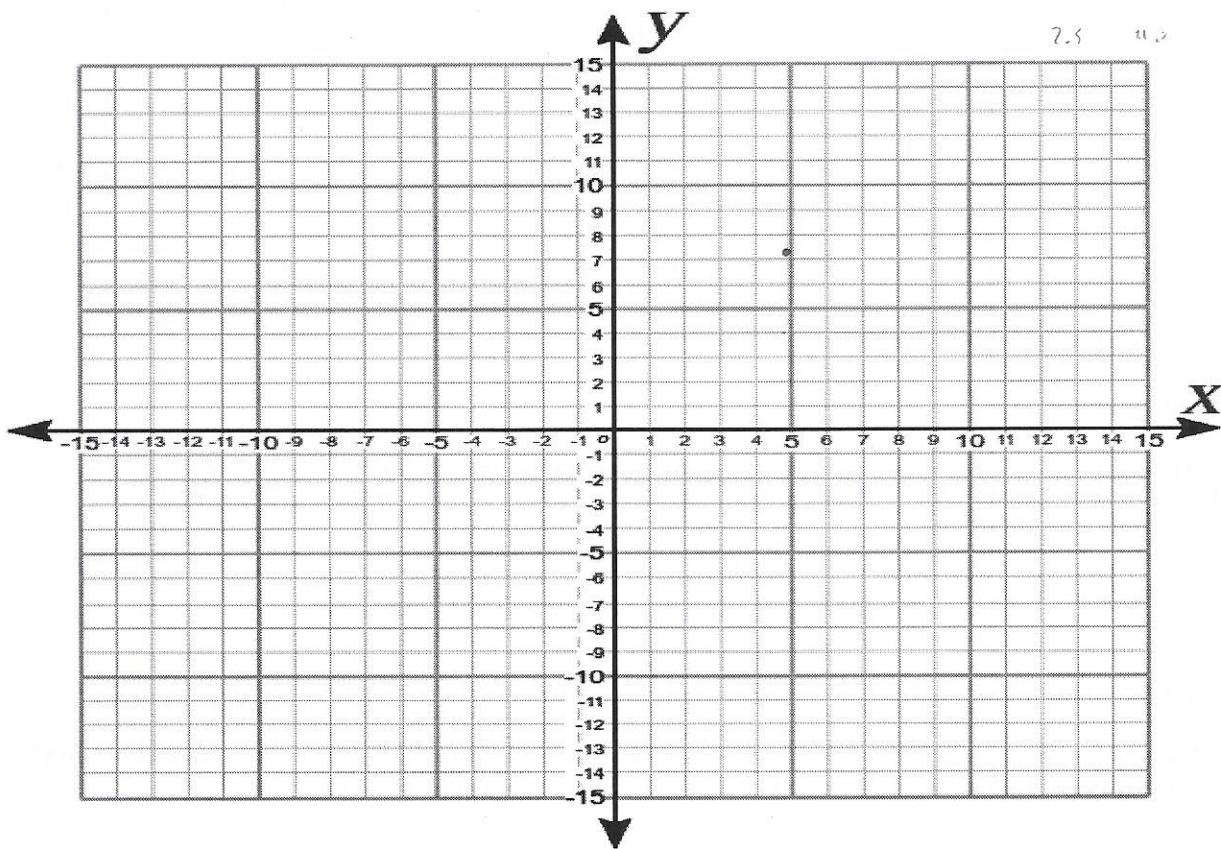


I =

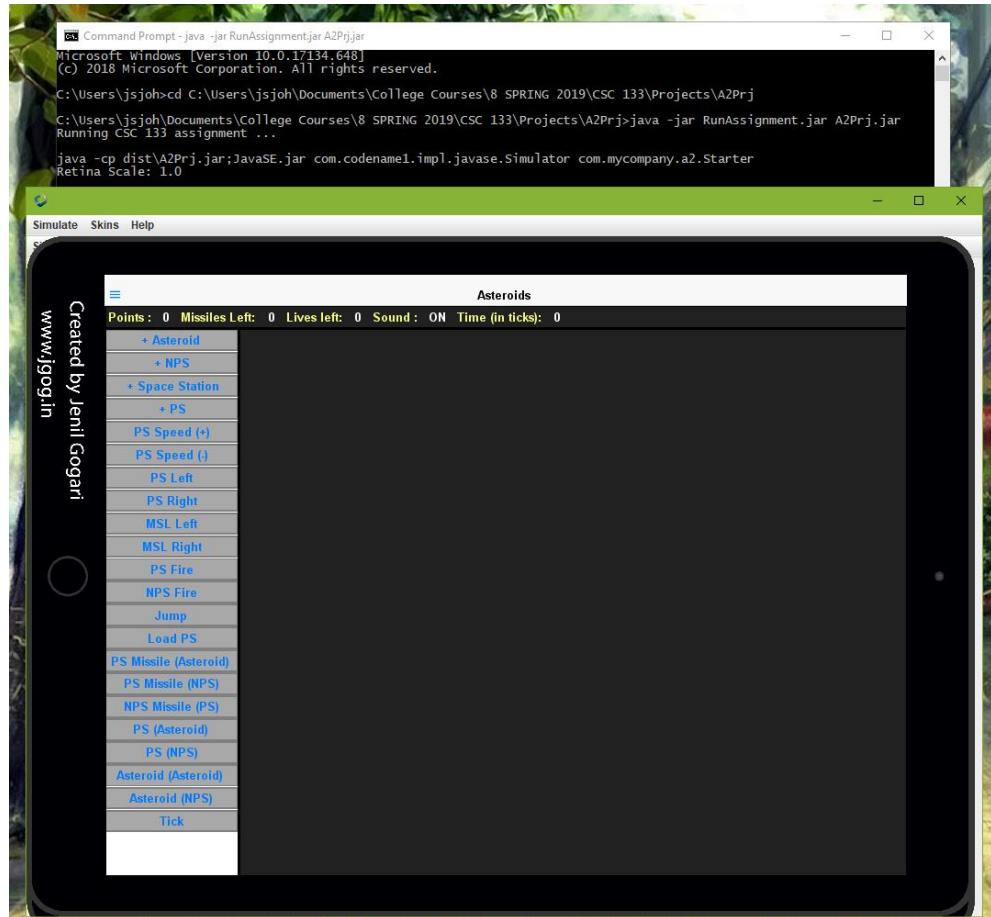
Show all the matrices computation work here: (10 points)

$$\begin{array}{l}
 \left[\begin{array}{c} 5+3 \\ 6-2 \\ 1 \end{array} \right] = \left[\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right] \times \left[\begin{array}{c} 0 \\ 10 \\ 1 \end{array} \right] + \left[\begin{array}{c} 3 \\ -2 \\ 1 \end{array} \right] \\
 \left[\begin{array}{c} 5+3 \\ 6-2 \\ 1 \end{array} \right] = \left[\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right] \times \left[\begin{array}{c} 5 \\ 6 \\ 1 \end{array} \right] + \left[\begin{array}{c} 3 \\ -2 \\ 1 \end{array} \right] \\
 \left[\begin{array}{c} 5+3 \\ 6-2 \\ 1 \end{array} \right] = \left[\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right] \times \left[\begin{array}{c} 5 \\ 6 \\ 1 \end{array} \right] + \left[\begin{array}{c} 3 \\ -2 \\ 1 \end{array} \right]
 \end{array}$$

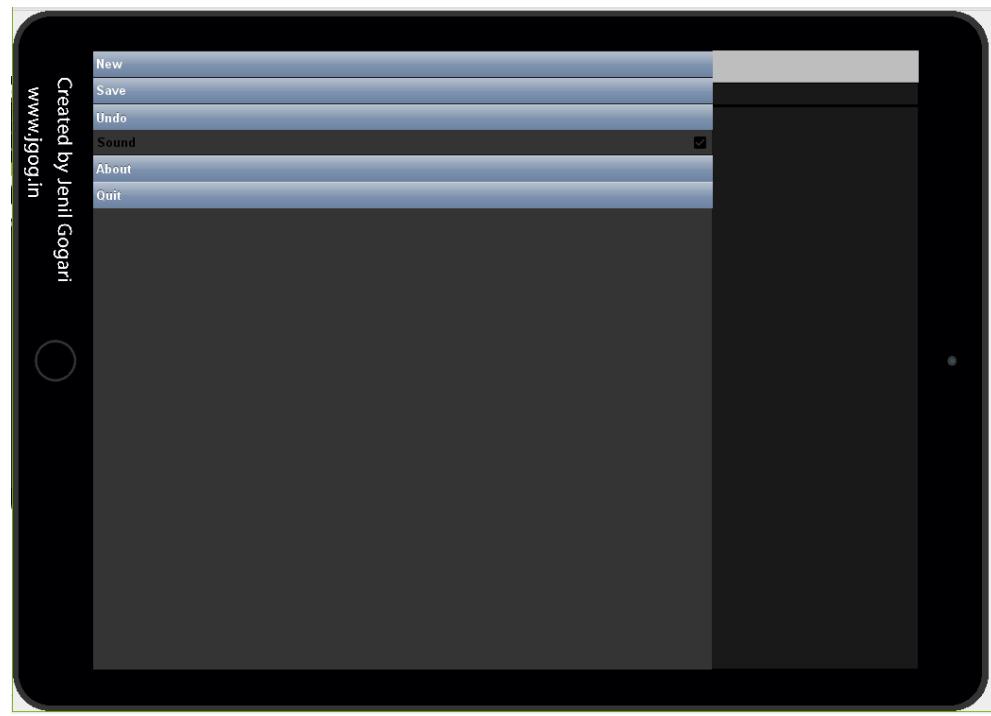
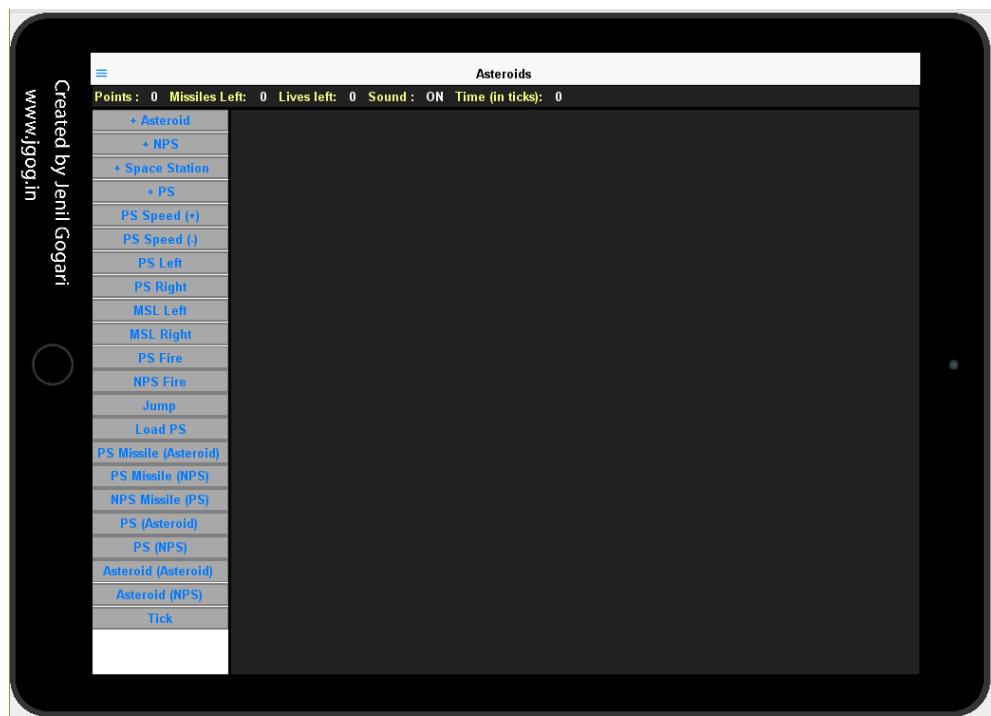
Draw your final transformation of this result here: (10 points)



Test Case 1: Opening the Program

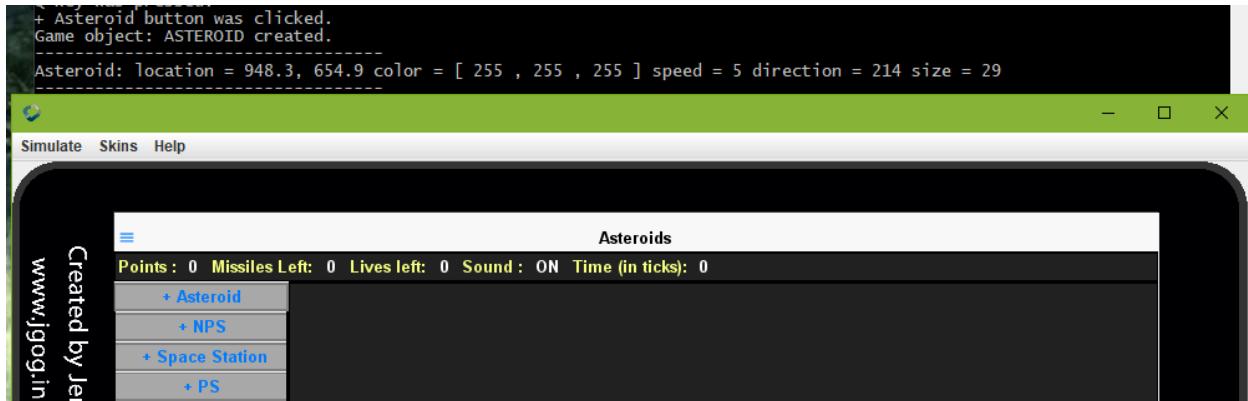


Test Case 2: GUI Displayed Correctly

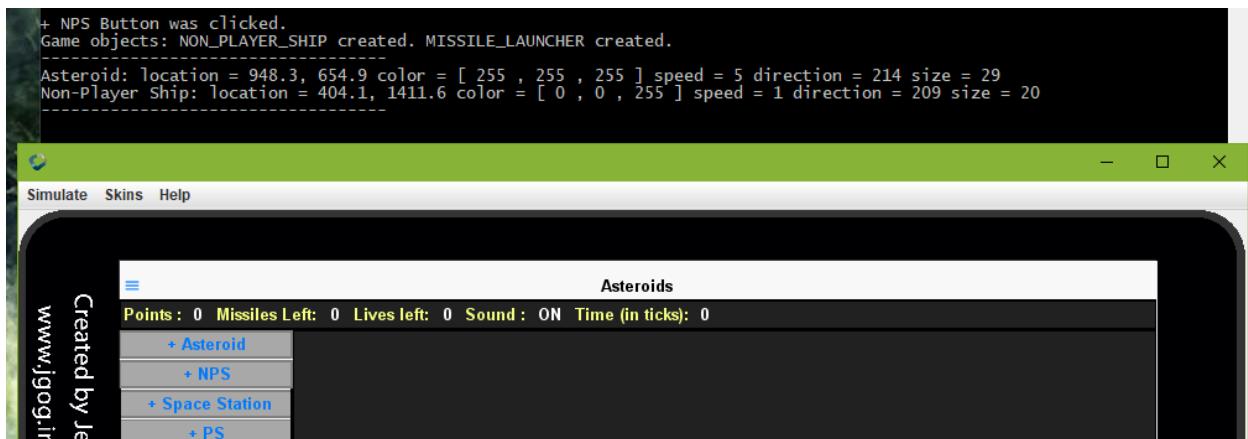


Test Case 3: Commands Performing Actions

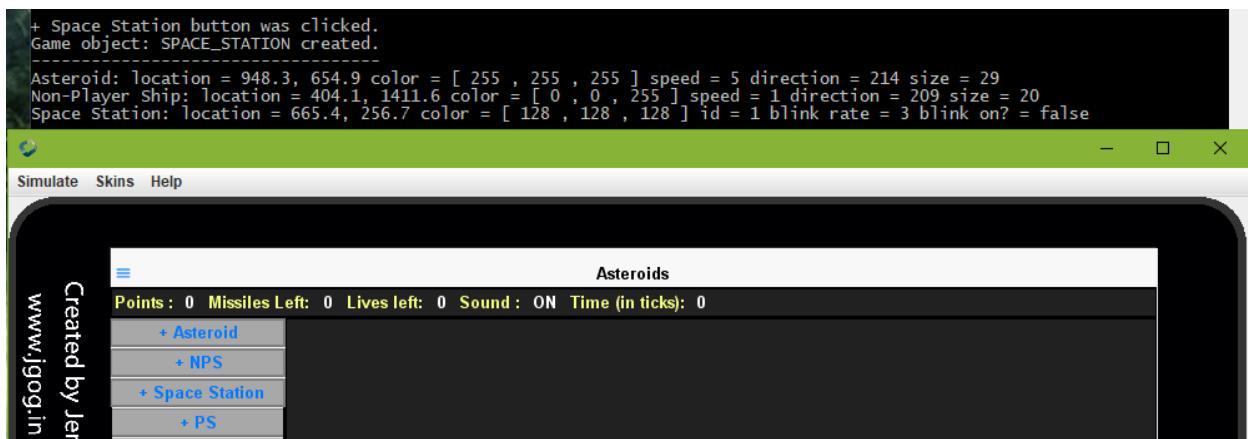
Asteroid button clicked:



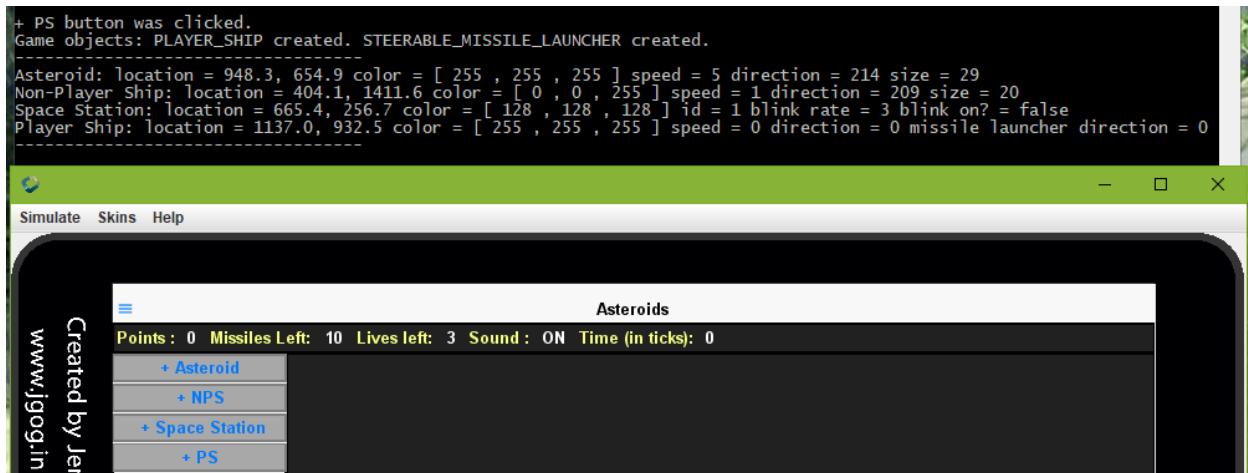
NPS button clicked:



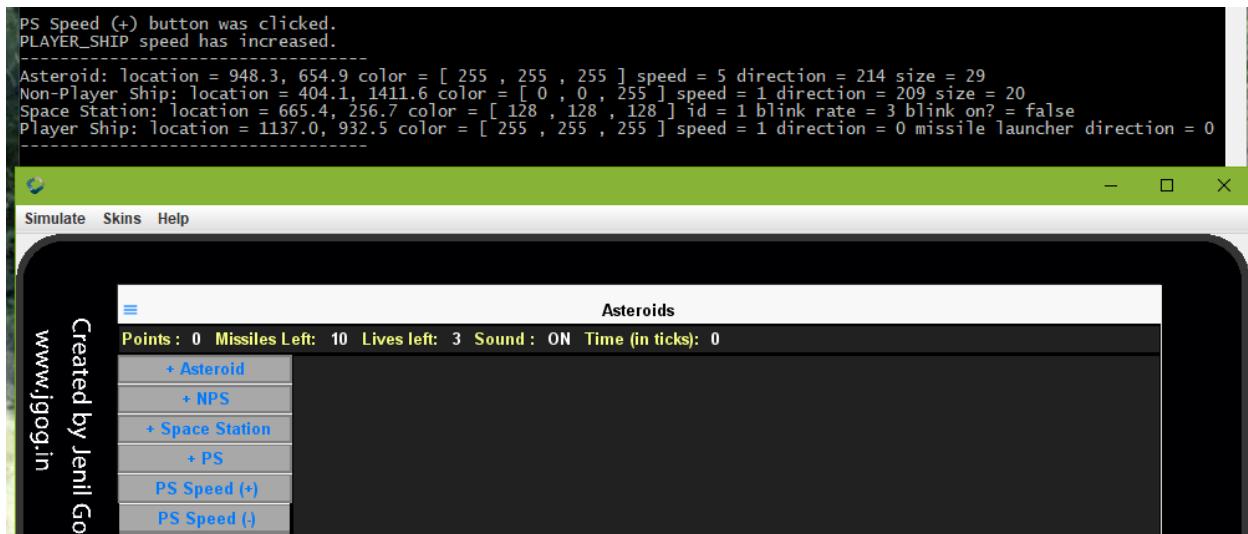
Space Station button clicked:



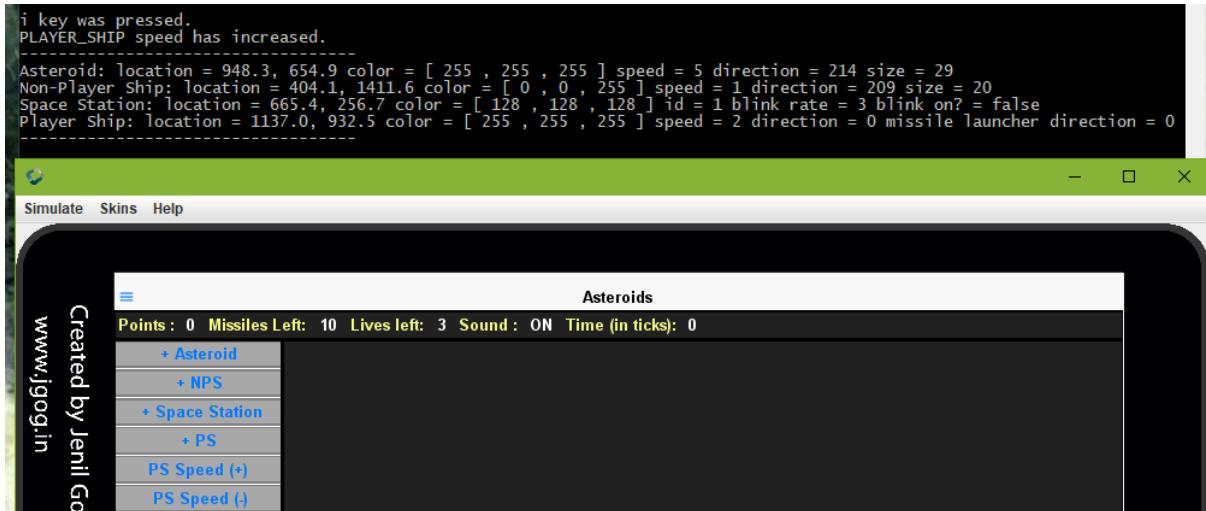
PS button clicked:



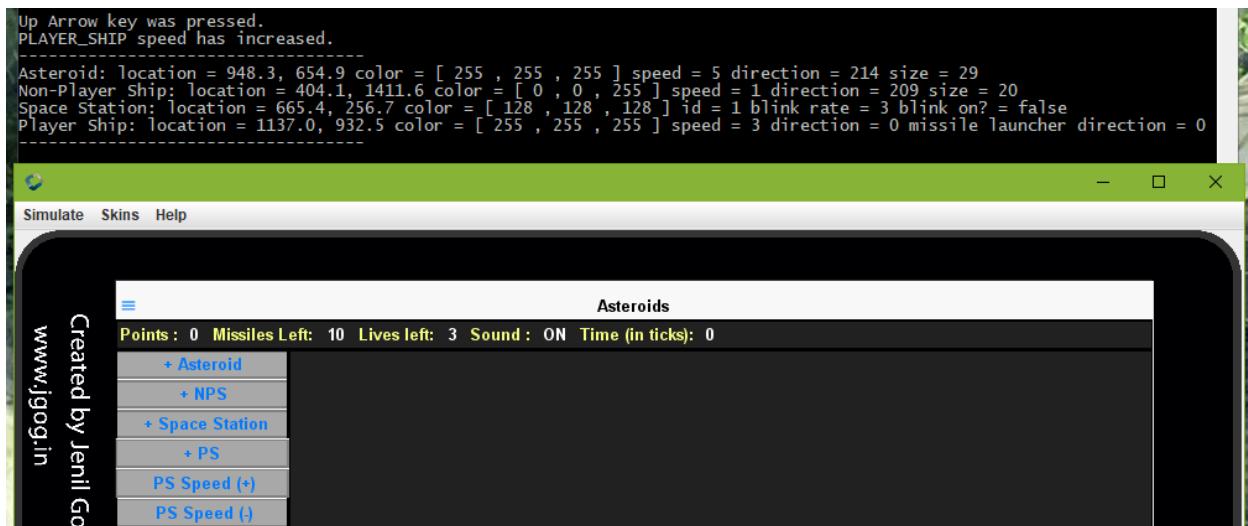
PS Speed (+) button clicked:



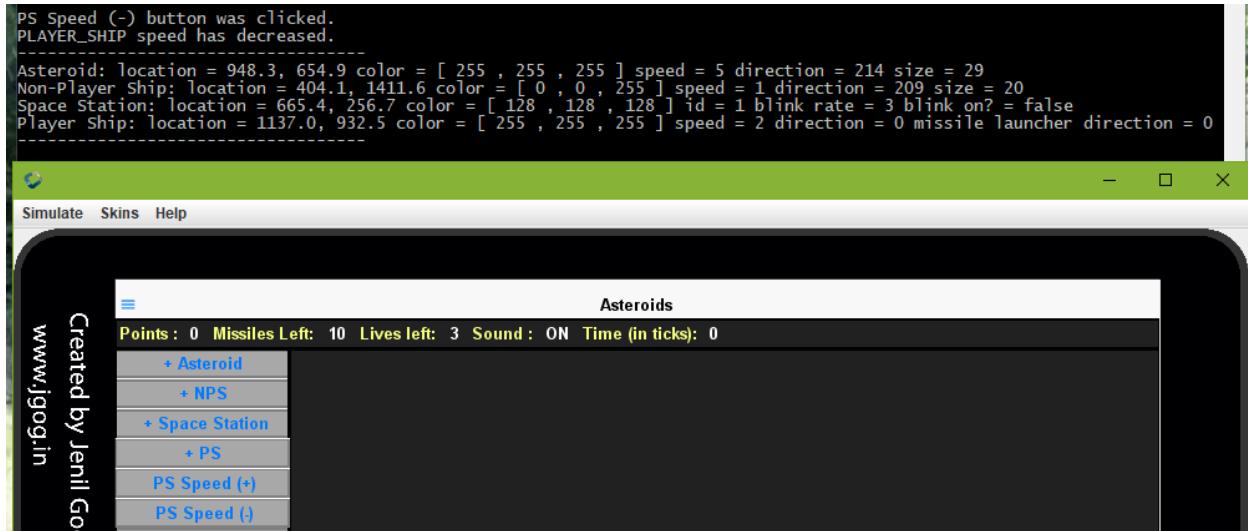
i key pressed:



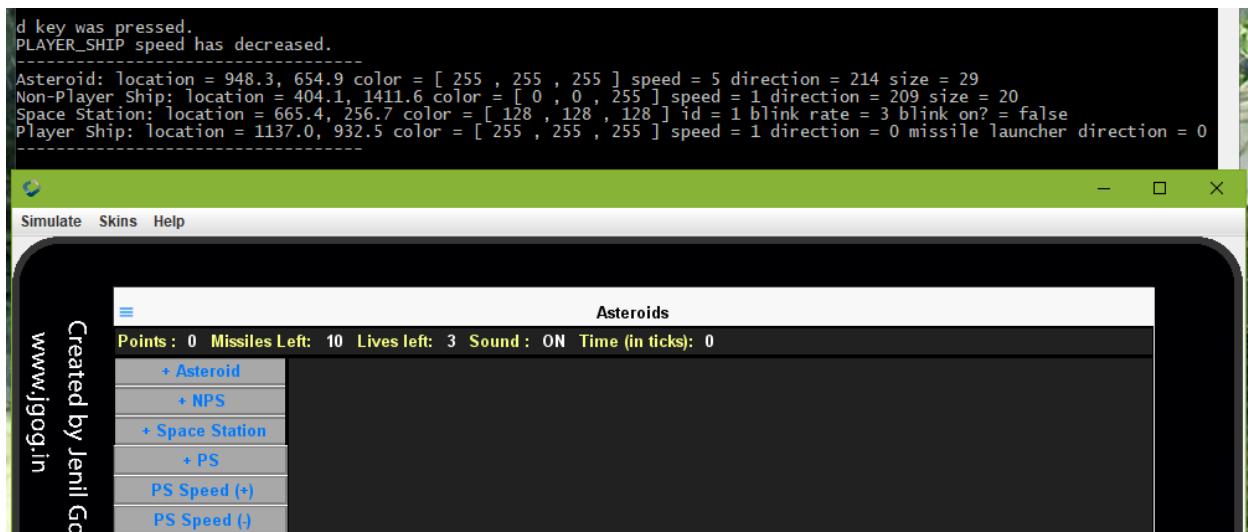
Up Arrow key pressed:



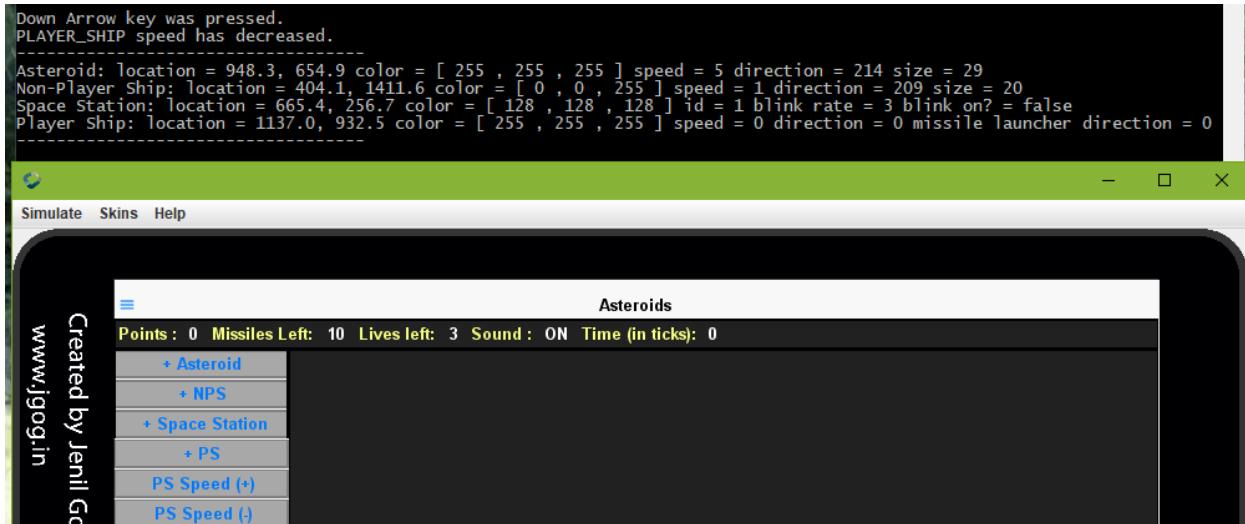
PS Speed (-) button clicked:



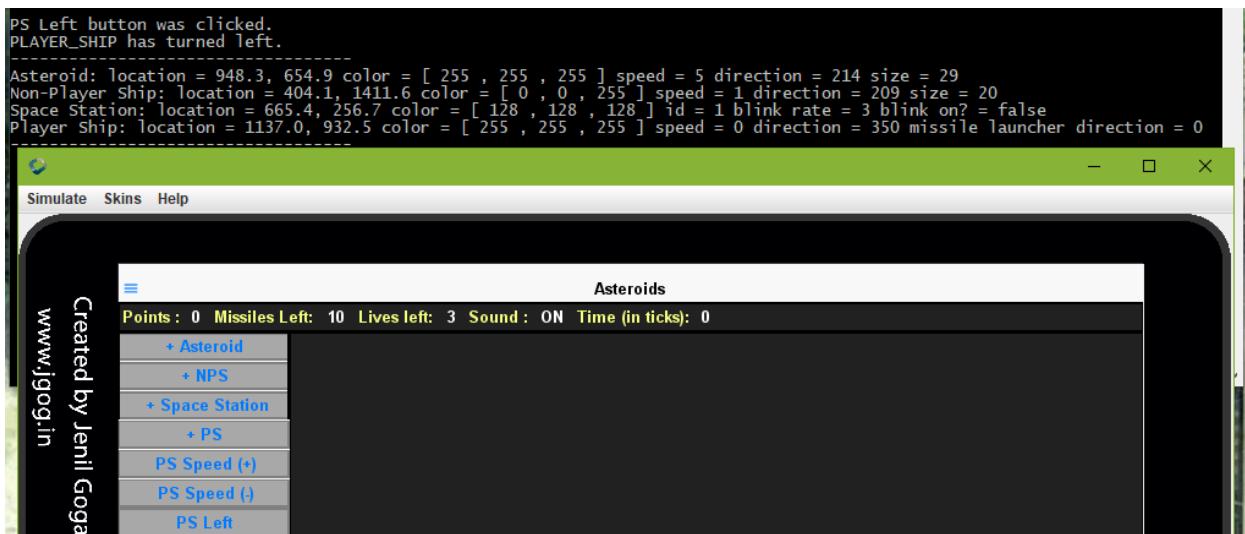
d key pressed:



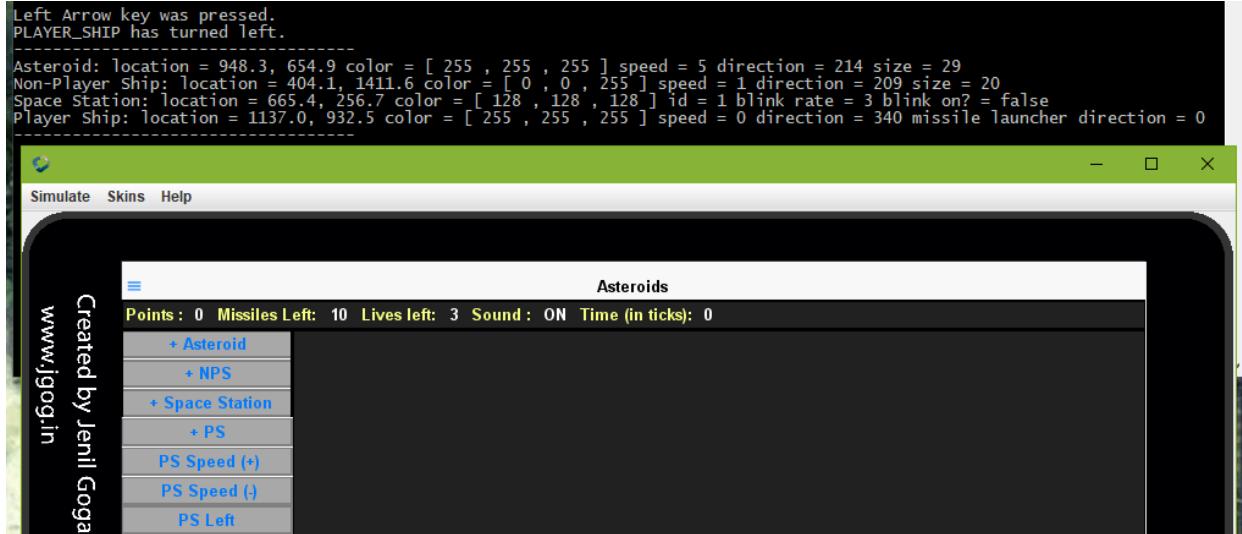
Down Arrow pressed:



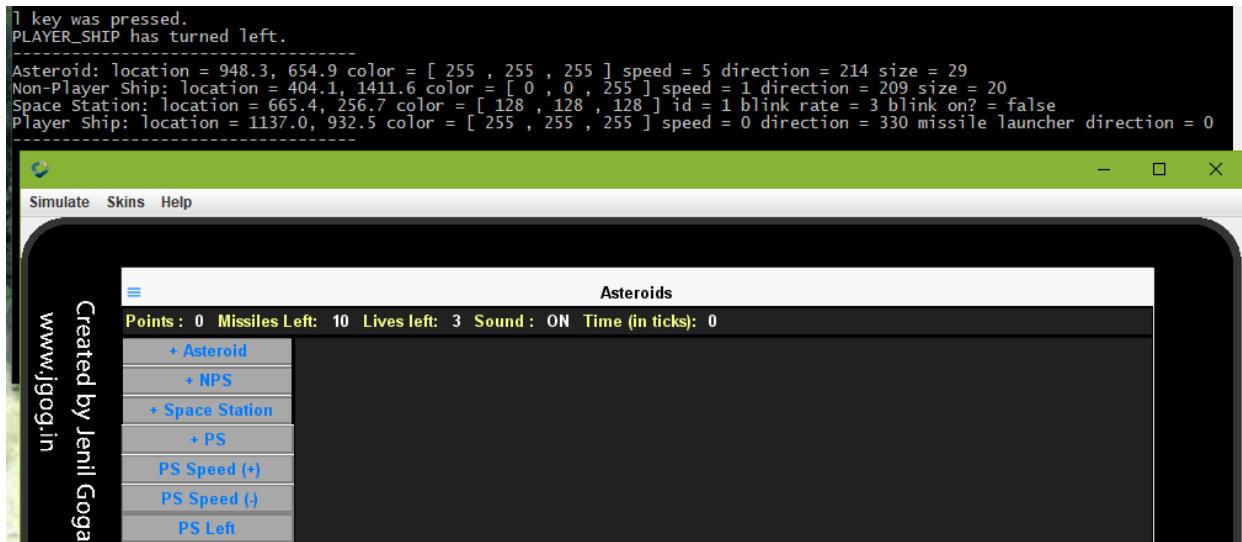
PS Left button clicked:



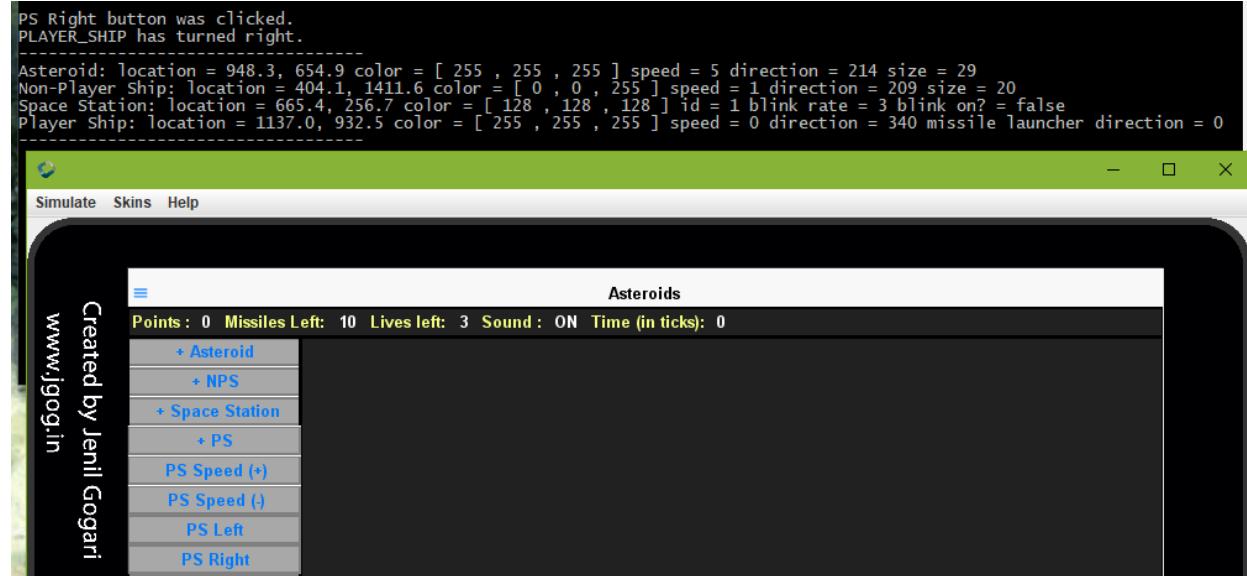
Left arrow key pressed:

```
Left Arrow key was pressed.  
PLAYER_SHIP has turned left.  
-----  
Asteroid: location = 948.3, 654.9 color = [ 255 , 255 , 255 ] speed = 5 direction = 214 size = 29  
Non-Player Ship: location = 404.1, 1411.6 color = [ 0 , 0 , 255 ] speed = 1 direction = 209 size = 20  
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false  
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 340 missile launcher direction = 0  
-----  

```

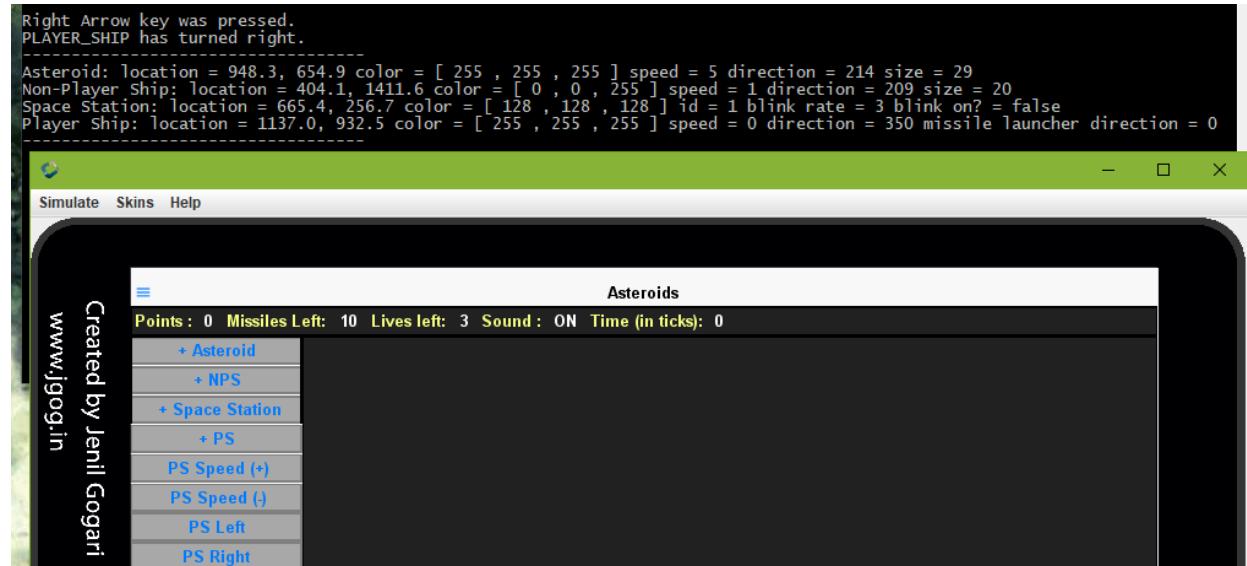
l key pressed:

```
l key was pressed.  
PLAYER_SHIP has turned left.  
-----  
Asteroid: location = 948.3, 654.9 color = [ 255 , 255 , 255 ] speed = 5 direction = 214 size = 29  
Non-Player Ship: location = 404.1, 1411.6 color = [ 0 , 0 , 255 ] speed = 1 direction = 209 size = 20  
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false  
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 330 missile launcher direction = 0  
-----  

```

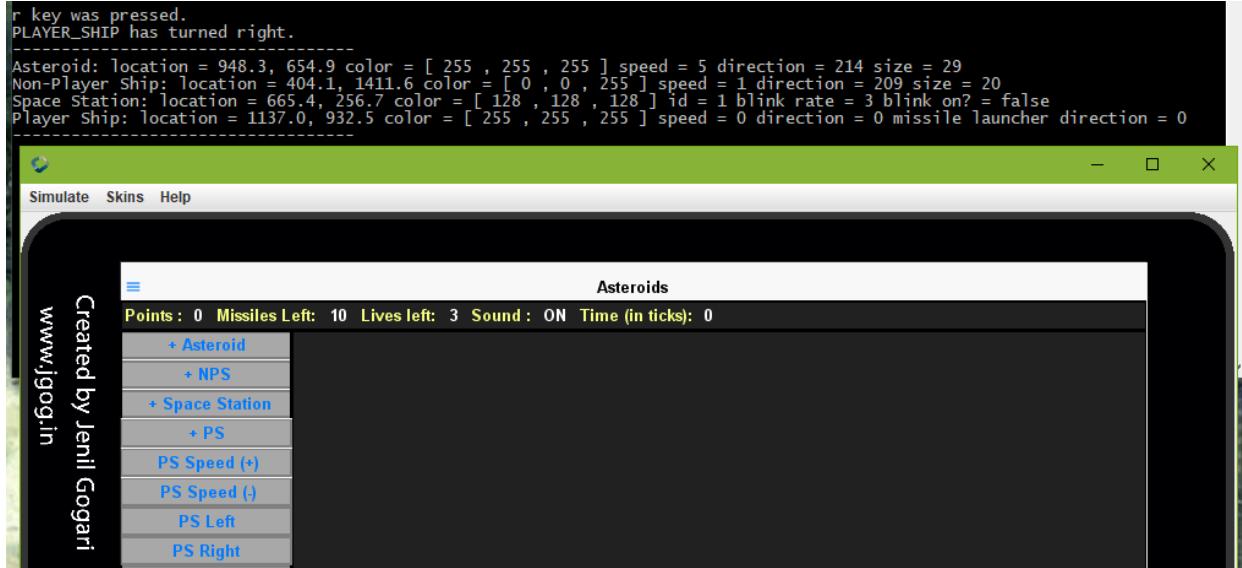
PS Right button clicked:

```
PS Right button was clicked.  
PLAYER_SHIP has turned right.  
-----  
Asteroid: location = 948.3, 654.9 color = [ 255 , 255 , 255 ] speed = 5 direction = 214 size = 29  
Non-Player Ship: location = 404.1, 1411.6 color = [ 0 , 0 , 255 ] speed = 1 direction = 209 size = 20  
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false  
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 340 missile launcher direction = 0  
-----  

```

Right Arrow key pressed:

```
Right Arrow key was pressed.  
PLAYER_SHIP has turned right.  
-----  
Asteroid: location = 948.3, 654.9 color = [ 255 , 255 , 255 ] speed = 5 direction = 214 size = 29  
Non-Player Ship: location = 404.1, 1411.6 color = [ 0 , 0 , 255 ] speed = 1 direction = 209 size = 20  
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false  
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 350 missile launcher direction = 0  
-----  

```

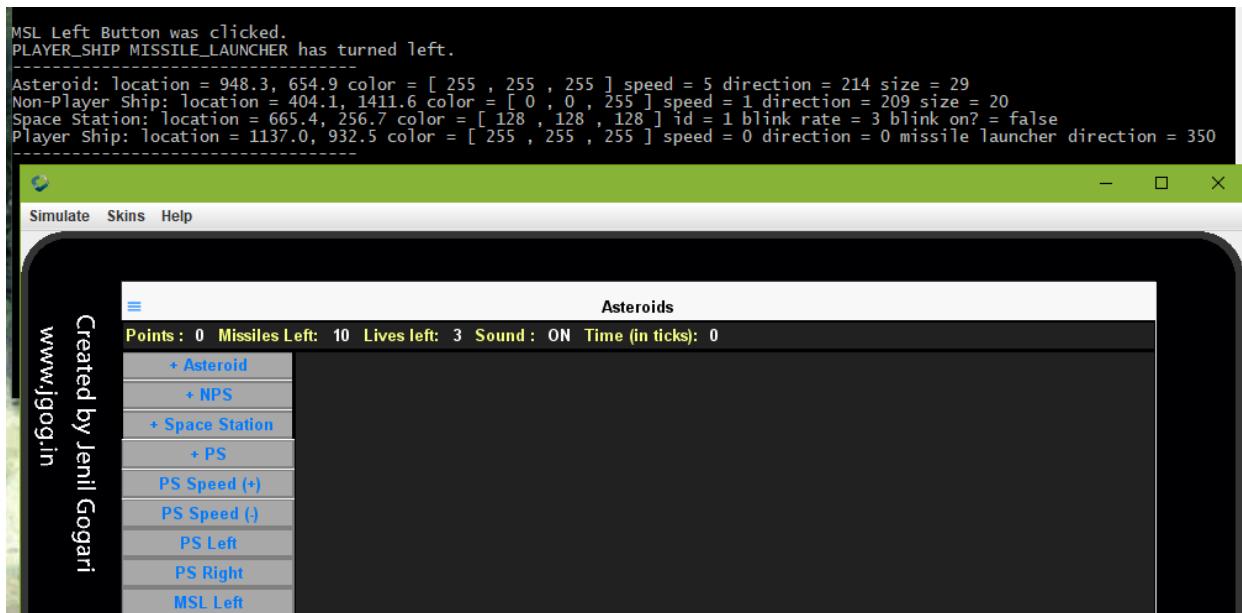
r key pressed:

```
r key was pressed.  
PLAYER_SHIP has turned right.  
-----  
Asteroid: location = 948.3, 654.9 color = [ 255 , 255 , 255 ] speed = 5 direction = 214 size = 29  
Non-Player Ship: location = 404.1, 1411.6 color = [ 0 , 0 , 255 ] speed = 1 direction = 209 size = 20  
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false  
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 0 missile launcher direction = 0  
-----  


Created by Jenil Gogari  
www.igog.in


```

MSL Left button clicked:

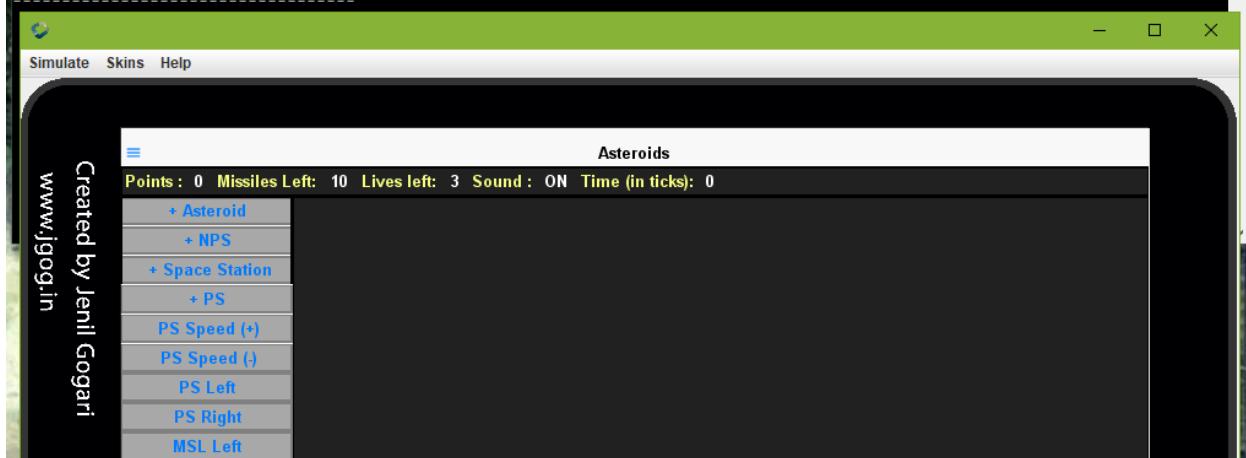
```
MSL Left Button was clicked.  
PLAYER_SHIP MISSILE_LAUNCHER has turned left.  
-----  
Asteroid: location = 948.3, 654.9 color = [ 255 , 255 , 255 ] speed = 5 direction = 214 size = 29  
Non-Player Ship: location = 404.1, 1411.6 color = [ 0 , 0 , 255 ] speed = 1 direction = 209 size = 20  
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false  
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 0 missile launcher direction = 350  
-----  


Created by Jenil Gogari  
www.igog.in


```

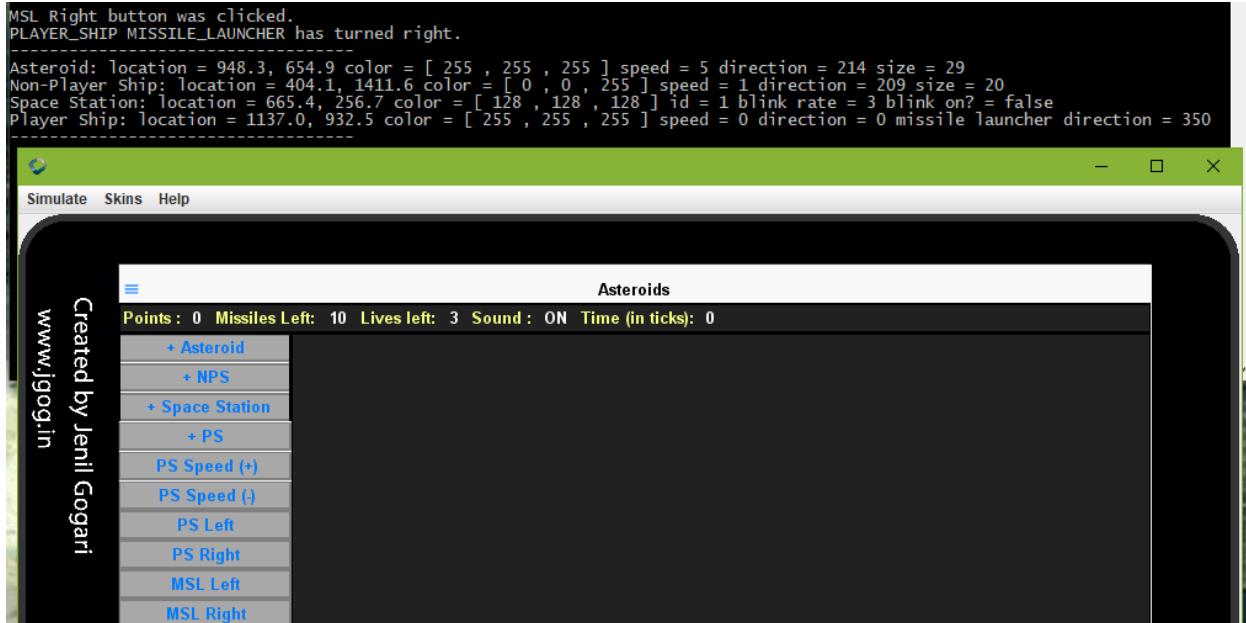
< key pressed:

```
< key was pressed.  
PLAYER_SHIP MISSILE_LAUNCHER has turned left.  
-----  
Asteroid: location = 948.3, 654.9 color = [ 255 , 255 , 255 ] speed = 5 direction = 214 size = 29  
Non-Player Ship: location = 404.1, 1411.6 color = [ 0 , 0 , 255 ] speed = 1 direction = 209 size = 20  
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false  
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 0 missile launcher direction = 340
```



MSL Right Button clicked:

```
MSL Right button was clicked.  
PLAYER_SHIP MISSILE_LAUNCHER has turned right.  
-----  
Asteroid: location = 948.3, 654.9 color = [ 255 , 255 , 255 ] speed = 5 direction = 214 size = 29  
Non-Player Ship: location = 404.1, 1411.6 color = [ 0 , 0 , 255 ] speed = 1 direction = 209 size = 20  
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false  
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 0 missile launcher direction = 350
```



> key pressed:

```
> key was pressed.  
PLAYER_SHIP MISSILE_LAUNCHER has turned right.  
-----  
Asteroid: location = 948.3, 654.9 color = [ 255 , 255 , 255 ] speed = 5 direction = 214 size = 29  
Non-Player Ship: location = 404.1, 1411.6 color = [ 0 , 0 , 255 ] speed = 1 direction = 209 size = 20  
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false  
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 0 missile launcher direction = 0  
-----  

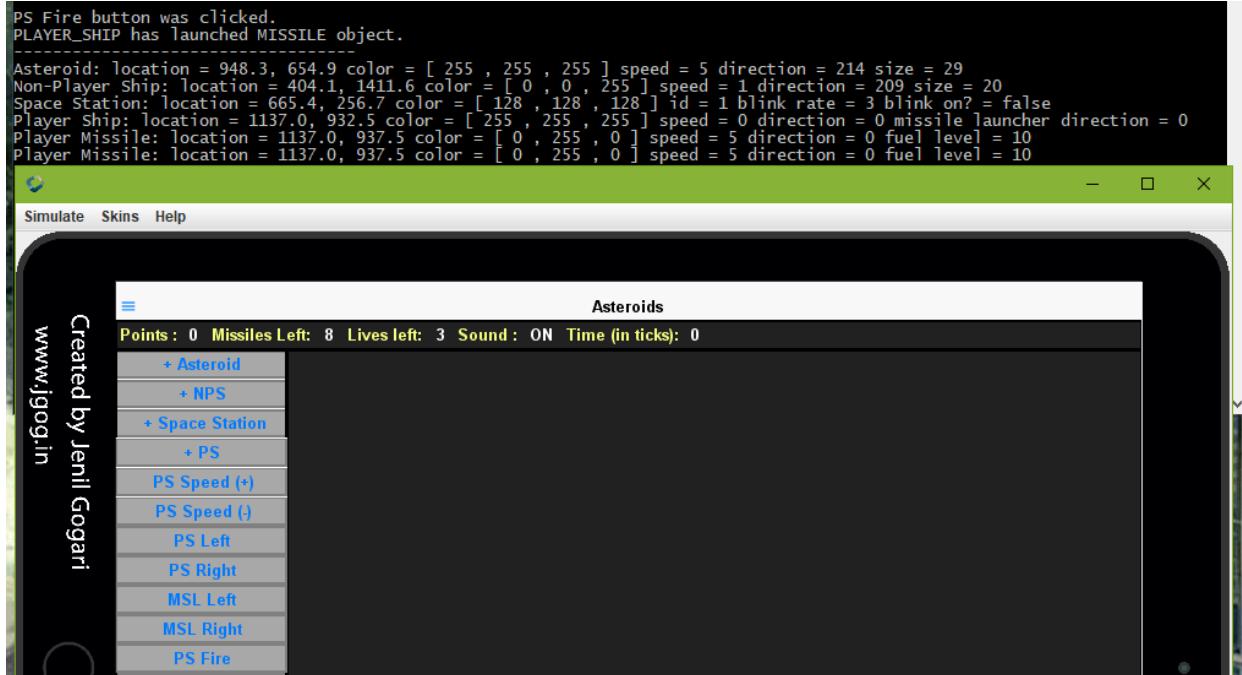
```

Spacebar key pressed:

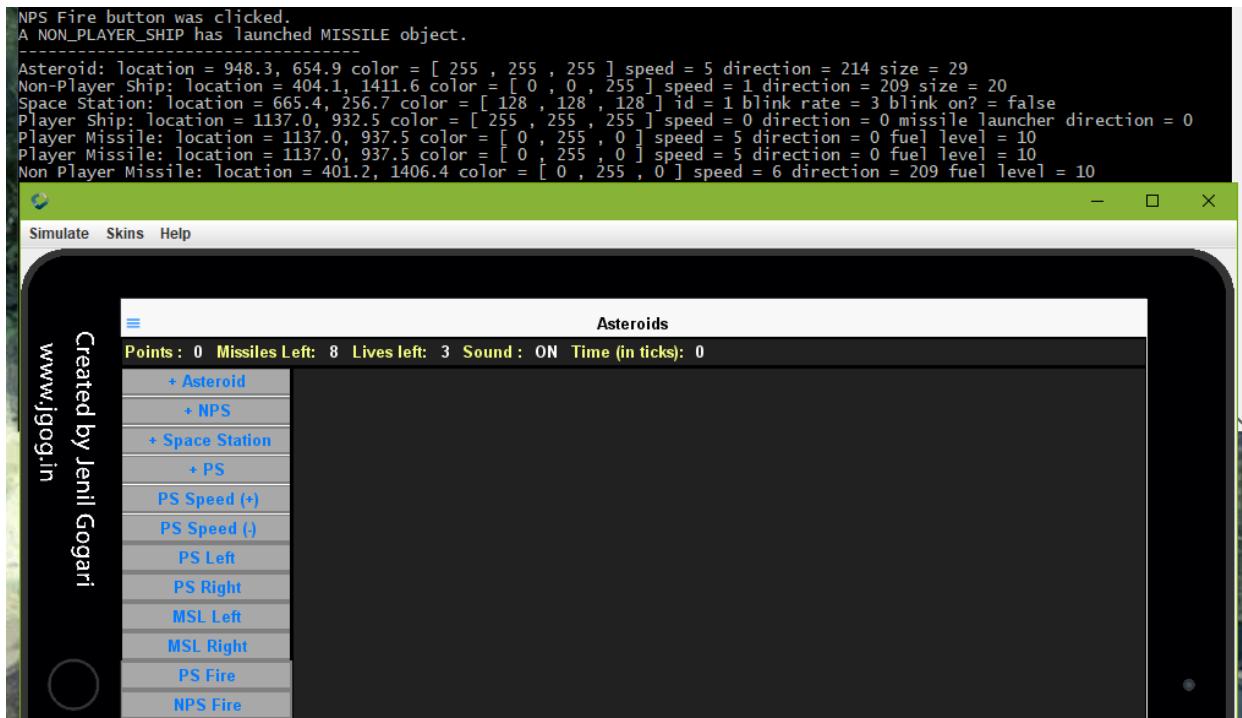
```
Spacebar key was pressed.  
PLAYER_SHIP has launched MISSILE object.  
-----  
Asteroid: location = 948.3, 654.9 color = [ 255 , 255 , 255 ] speed = 5 direction = 214 size = 29  
Non-Player Ship: location = 404.1, 1411.6 color = [ 0 , 0 , 255 ] speed = 1 direction = 209 size = 20  
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false  
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 0 missile launcher direction = 0  
Player Missile: location = 1137.0, 937.5 color = [ 0 , 255 , 0 ] speed = 5 direction = 0 fuel level = 10  
-----  

```

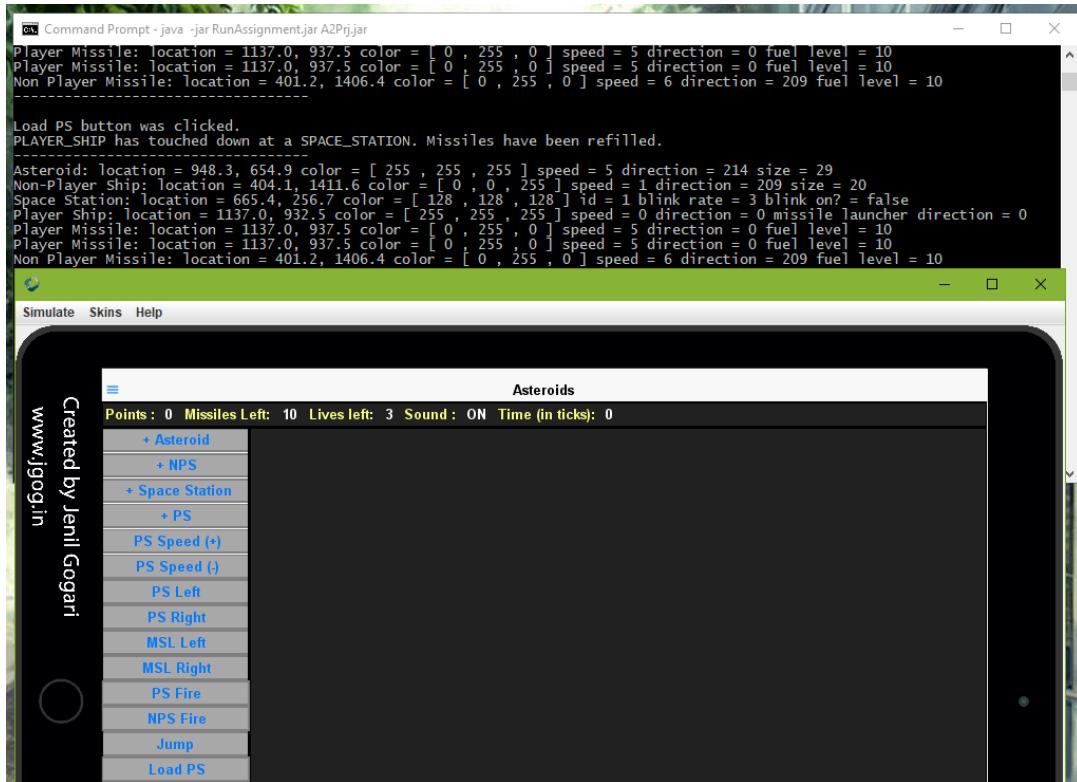
PS Fire Button clicked:



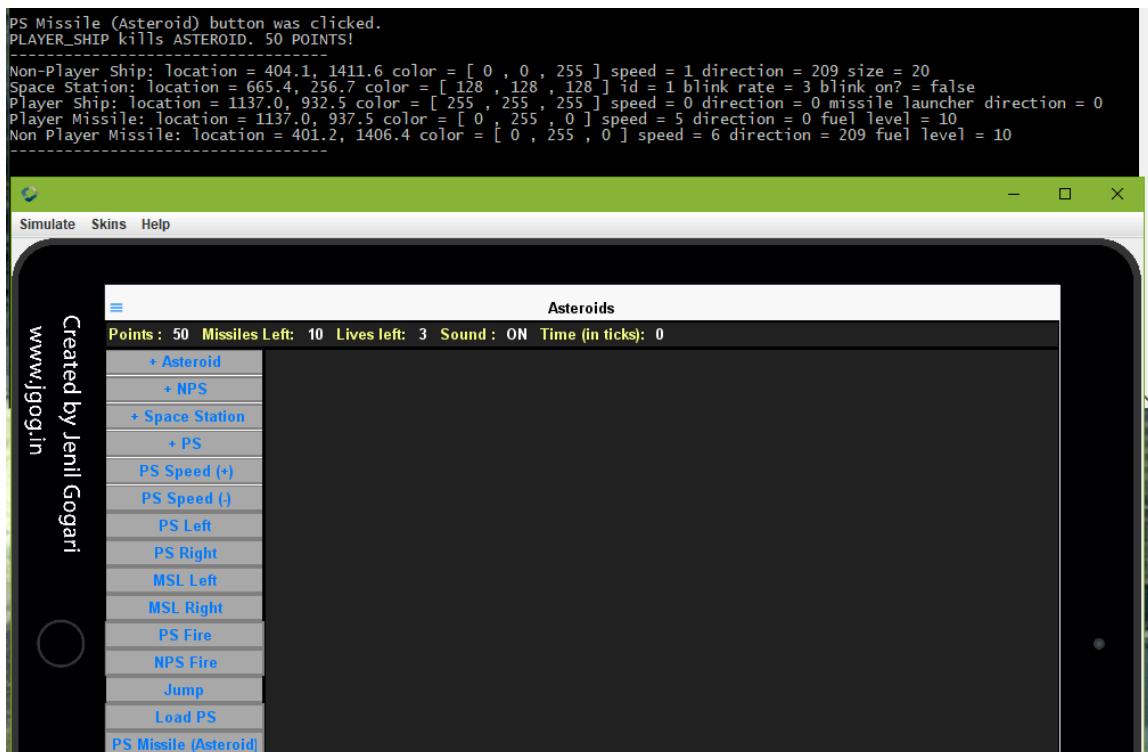
NPS Fire button clicked:



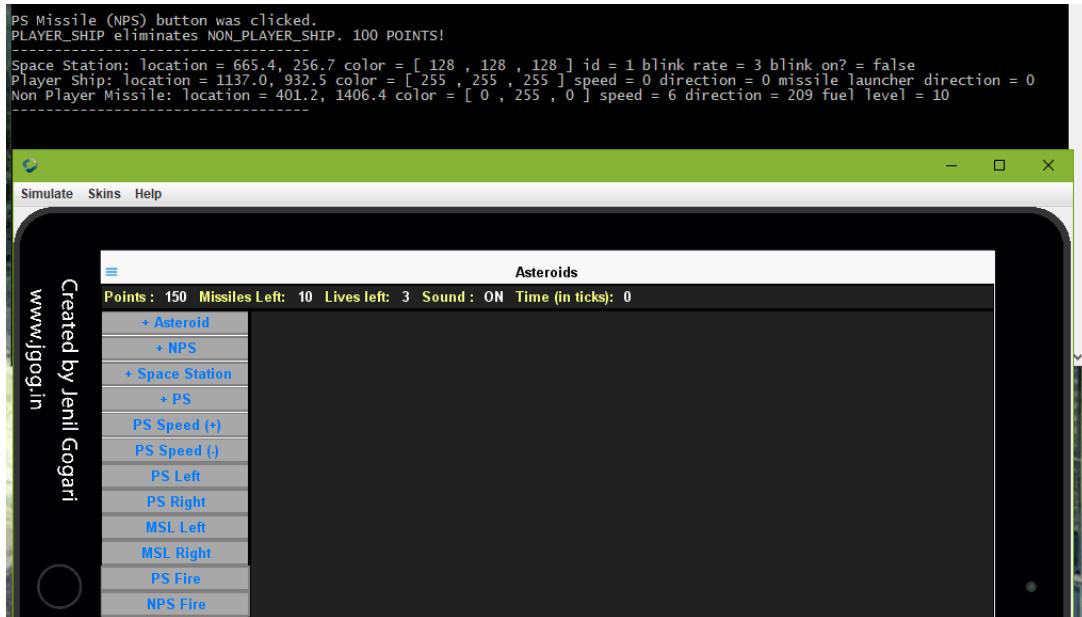
Load PS button clicked:



PS Missile (Asteroid) button clicked:

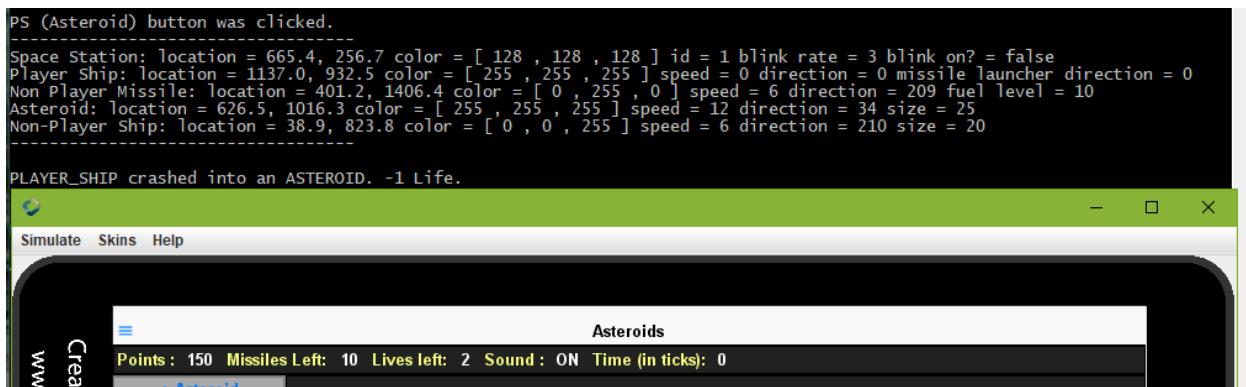


PS Missile (NPS) button clicked:

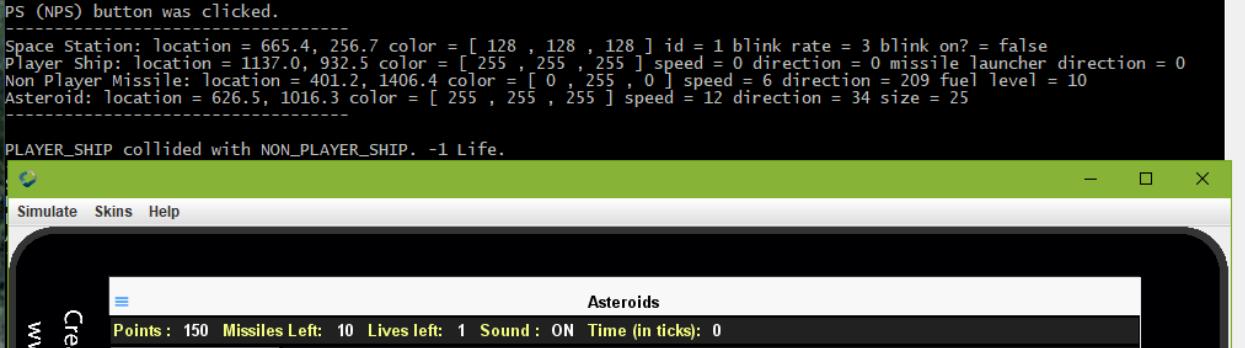


PS (Asteroid) Button clicked:

```
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 0 missile launcher direction = 0
Non Player Missile: location = 401.2, 1406.4 color = [ 0 , 255 , 0 ] speed = 6 direction = 209 fuel level = 10
Asteroid: location = 562.2, 390.7 color = [ 255 , 255 , 255 ] speed = 15 direction = 294 size = 29
Asteroid: location = 626.5, 1016.3 color = [ 255 , 255 , 255 ] speed = 12 direction = 34 size = 25
Non-Player Ship: location = 38.9, 823.8 color = [ 0 , 0 , 255 ] speed = 6 direction = 210 size = 20
```



PS (NPS) Button clicked:

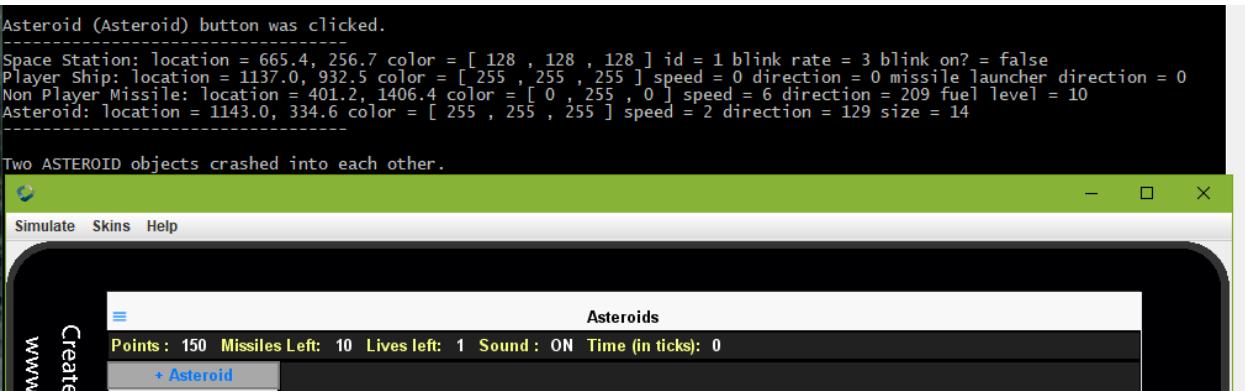
```
PS (NPS) button was clicked.  
-----  
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false  
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 0 missile launcher direction = 0  
Non Player Missile: location = 401.2, 1406.4 color = [ 0 , 255 , 0 ] speed = 6 direction = 209 fuel level = 10  
Asteroid: location = 626.5, 1016.3 color = [ 255 , 255 , 255 ] speed = 12 direction = 34 size = 25  
  
PLAYER_SHIP collided with NON_PLAYER_SHIP. -1 Life.  


The screenshot shows a Windows-style window titled "Asteroids". The menu bar includes "Simulate", "Skins", and "Help". Below the menu is a toolbar with icons for "Create" and "www". The main game area shows a black space background with several white and grey asteroid shapes. A small white player ship is visible. At the bottom of the screen, there is a status bar displaying: Points : 150 Missiles Left: 10 Lives left: 1 Sound : ON Time (in ticks): 0.


```

Asteroid (Asteroid) button clicked:

```
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false  
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 0 missile launcher direction = 0  
Non Player Missile: location = 401.2, 1406.4 color = [ 0 , 255 , 0 ] speed = 6 direction = 209 fuel level = 10  
Asteroid: location = 626.5, 1016.3 color = [ 255 , 255 , 255 ] speed = 12 direction = 34 size = 25  
Asteroid: location = 1060.4, 228.6 color = [ 255 , 255 , 255 ] speed = 14 direction = 292 size = 27  
Asteroid: location = 1143.0, 334.6 color = [ 255 , 255 , 255 ] speed = 2 direction = 129 size = 14  
-----
```

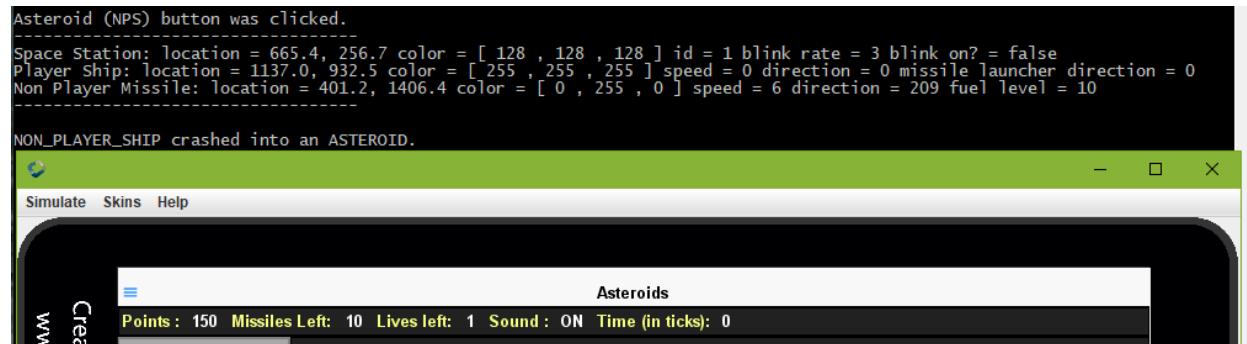
```
Asteroid (Asteroid) button was clicked.  
-----  
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false  
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 0 missile launcher direction = 0  
Non Player Missile: location = 401.2, 1406.4 color = [ 0 , 255 , 0 ] speed = 6 direction = 209 fuel level = 10  
Asteroid: location = 1143.0, 334.6 color = [ 255 , 255 , 255 ] speed = 2 direction = 129 size = 14  
  
Two ASTEROID objects crashed into each other.  


The screenshot shows a Windows-style window titled "Asteroids". The menu bar includes "Simulate", "Skins", and "Help". Below the menu is a toolbar with icons for "Create" and "www". The main game area shows a black space background with several white and grey asteroid shapes. Two asteroids have collided and exploded into smaller pieces. At the bottom of the screen, there is a status bar displaying: Points : 150 Missiles Left: 10 Lives left: 1 Sound : ON Time (in ticks): 0. A message "+ Asteroid" is displayed at the bottom of the status bar.


```

Asteroid (NPS) button clicked:

```
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 0 missile launcher direction = 0
Non Player Missile: location = 401.2, 1406.4 color = [ 0 , 255 , 0 ] speed = 6 direction = 209 fuel level = 10
Asteroid: location = 1143.0, 334.6 color = [ 255 , 255 , 255 ] speed = 2 direction = 129 size = 14
Non-Player Ship: location = 454.1, 1486.0 color = [ 0 , 0 , 255 ] speed = 7 direction = 65 size = 10
```



Tick button clicked:

```
Tick button was clicked.
Time has passed.
```

```
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 0 direction = 0 missile launcher direction = 0
Non Player Missile: location = 398.3, 1401.1 color = [ 0 , 255 , 0 ] speed = 6 direction = 209 fuel level = 9
```

PS Speed (+) button was clicked.
PLAYER_SHIP speed has increased.

```
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false
```

```
Points : 150 Missiles Left: 10 Lives left: 1 Sound : ON Time (in ticks): 2
```

+ Asteroid

```
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false
Player Ship: location = 1137.0, 935.5 color = [ 255 , 255 , 255 ] speed = 3 direction = 0 missile launcher direction = 0
Non Player Missile: location = 395.3, 1395.9 color = [ 0 , 255 , 0 ] speed = 6 direction = 209 fuel level = 9
```

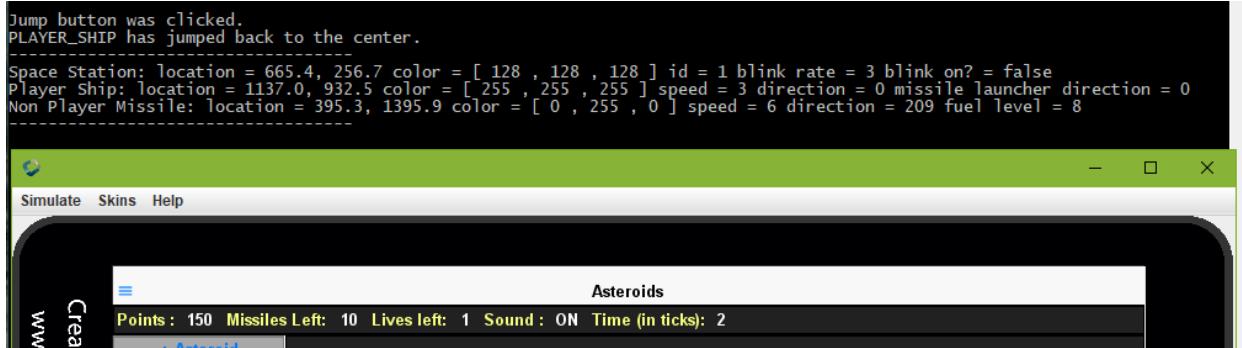
PS Speed (+) button was clicked.
PLAYER_SHIP speed has increased.

```
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false
Player Ship: location = 1137.0, 935.5 color = [ 255 , 255 , 255 ] speed = 3 direction = 0 missile launcher direction = 0
Non Player Missile: location = 395.3, 1395.9 color = [ 0 , 255 , 0 ] speed = 6 direction = 209 fuel level = 8
```

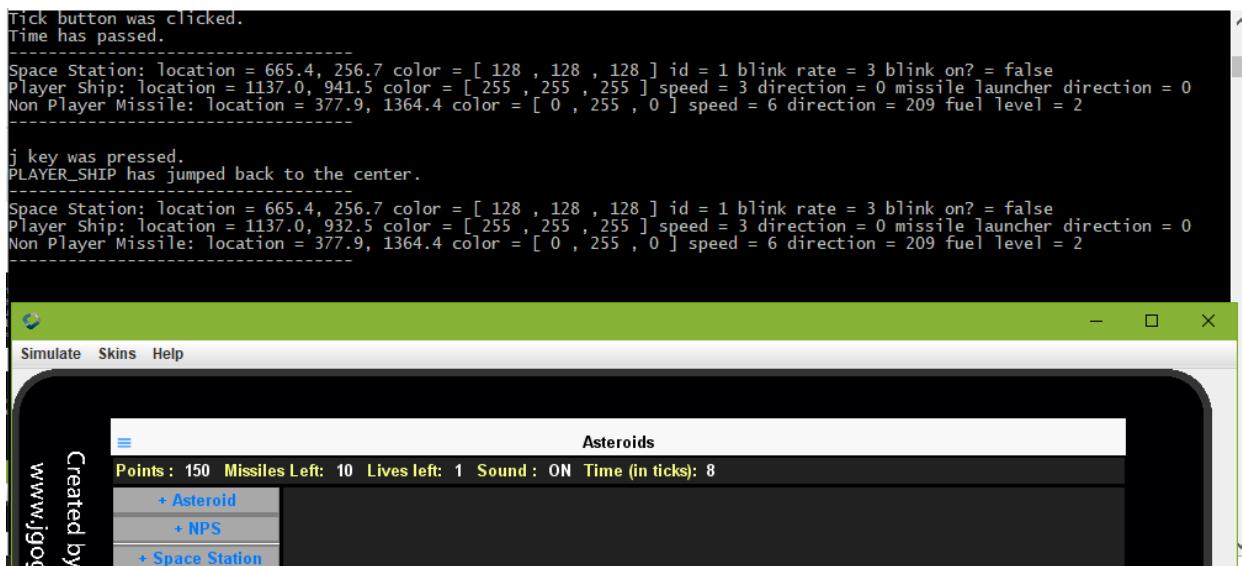
Tick button was clicked.
Time has passed.

```
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false
Player Ship: location = 1137.0, 935.5 color = [ 255 , 255 , 255 ] speed = 3 direction = 0 missile launcher direction = 0
Non Player Missile: location = 395.3, 1395.9 color = [ 0 , 255 , 0 ] speed = 6 direction = 209 fuel level = 8
```

Jump button clicked:



j key pressed:



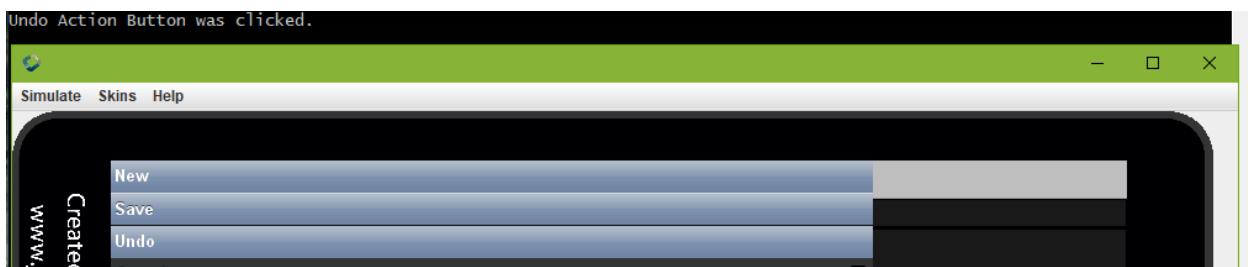
New Button clicked:



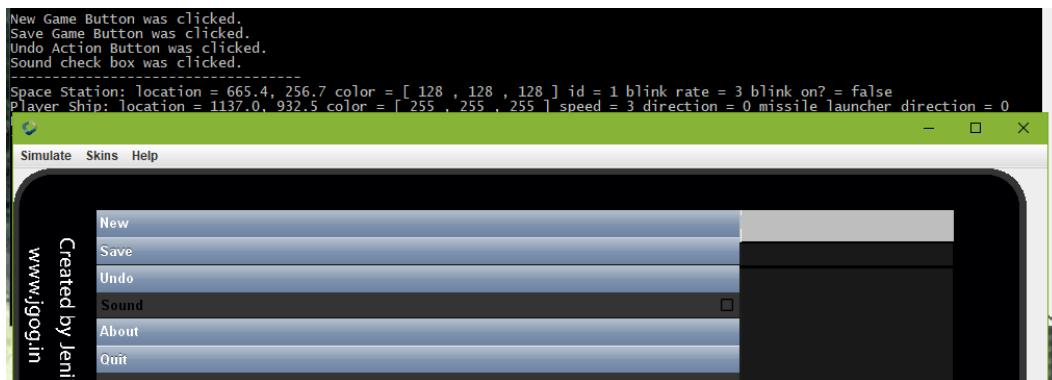
Save Button clicked:



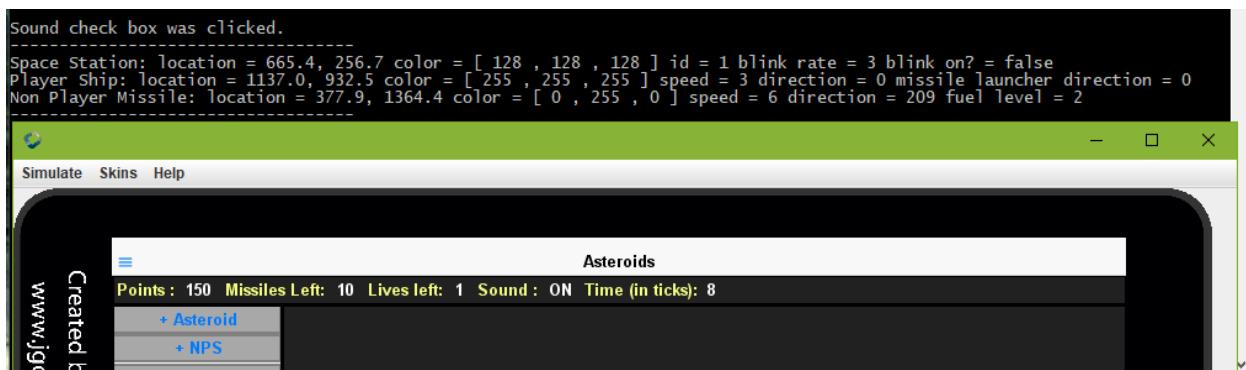
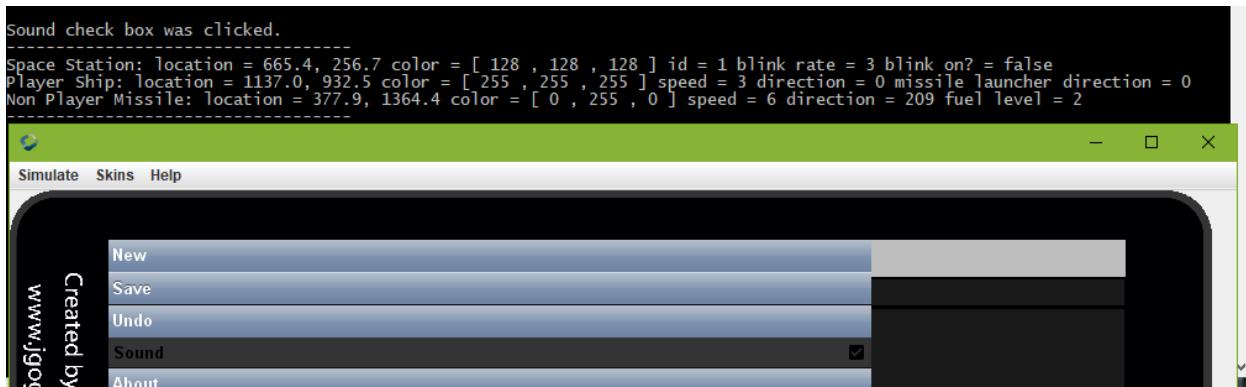
Undo Button clicked:



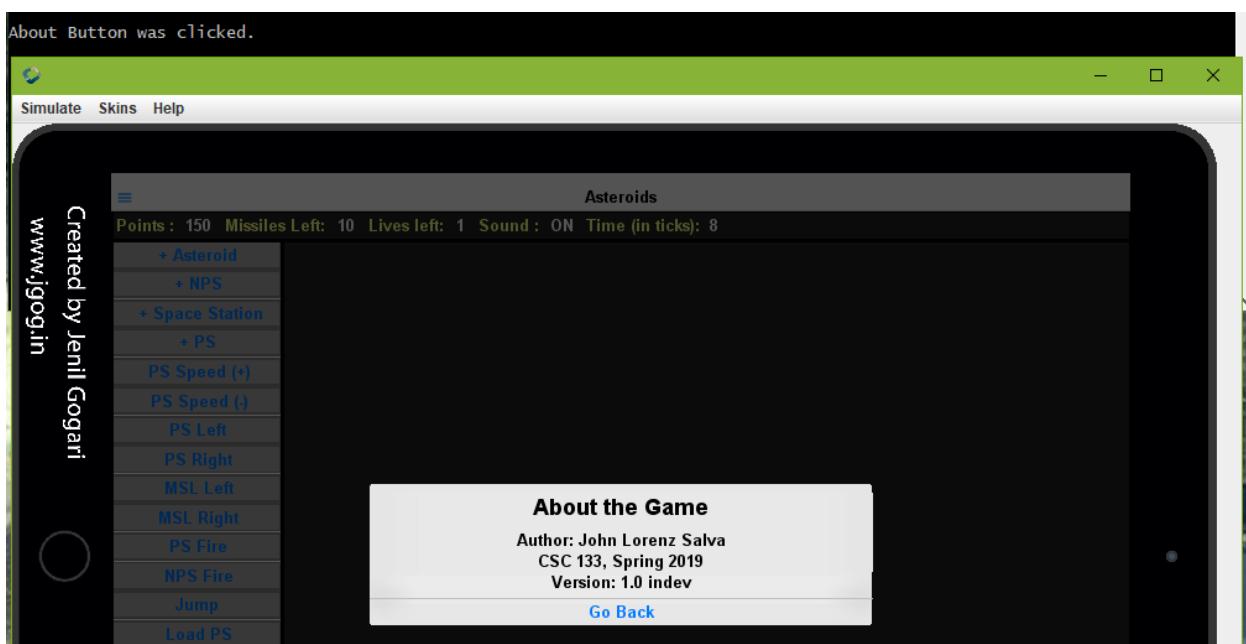
Sound Check Box clicked OFF:



Sound Check Box clicked ON:

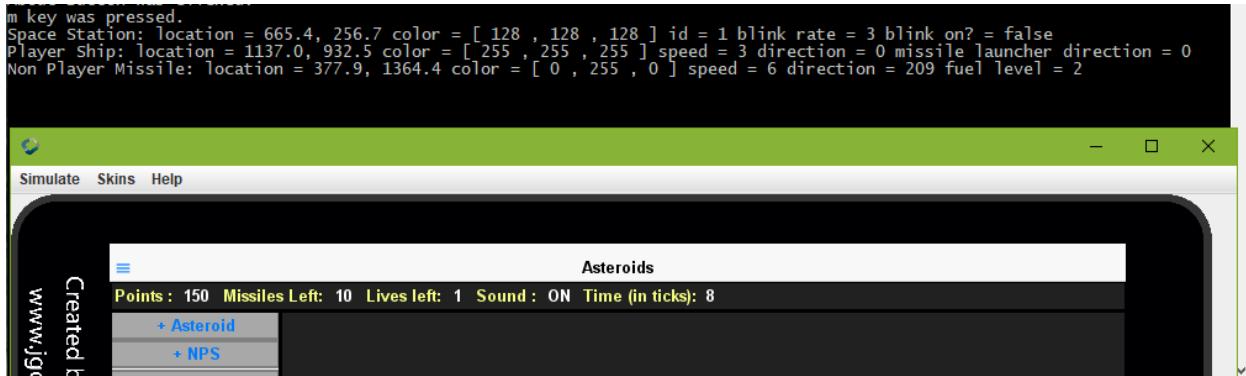


About Button clicked:



m key pressed:

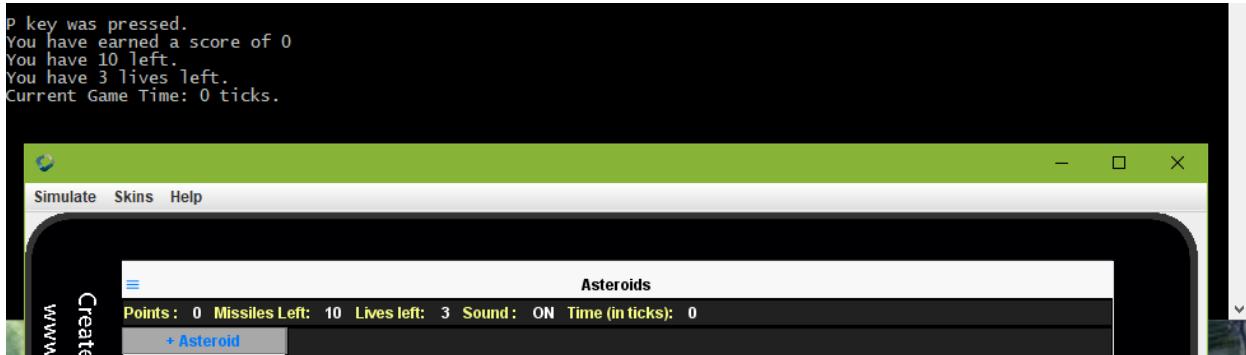
```
m key was pressed.  
Space Station: location = 665.4, 256.7 color = [ 128 , 128 , 128 ] id = 1 blink rate = 3 blink on? = false  
Player Ship: location = 1137.0, 932.5 color = [ 255 , 255 , 255 ] speed = 3 direction = 0 missile launcher direction = 0  
Non Player Missile: location = 377.9, 1364.4 color = [ 0 , 255 , 0 ] speed = 6 direction = 209 fuel level = 2
```



The screenshot shows a Java application window titled "Asteroids". The status bar at the bottom displays the following information: Points: 150 Missiles Left: 10 Lives left: 1 Sound: ON Time (in ticks): 8. There are two buttons below the status bar: "+ Asteroid" and "+ NPS". The main game area is visible in the background.

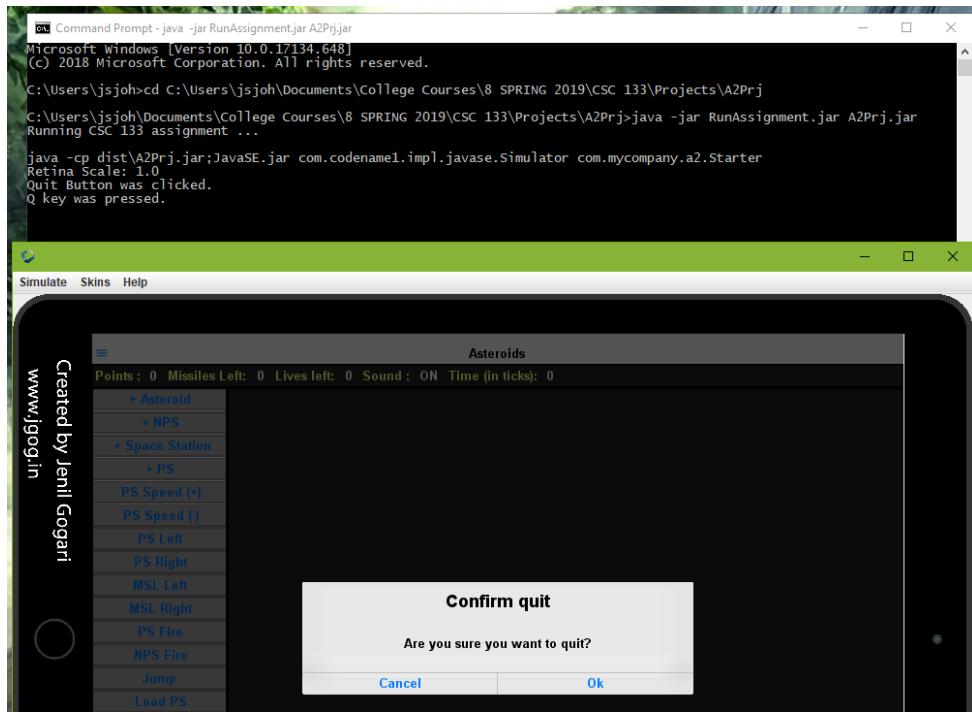
P key pressed:

```
p key was pressed.  
You have earned a score of 0  
You have 10 left.  
You have 3 lives left.  
Current Game Time: 0 ticks.
```

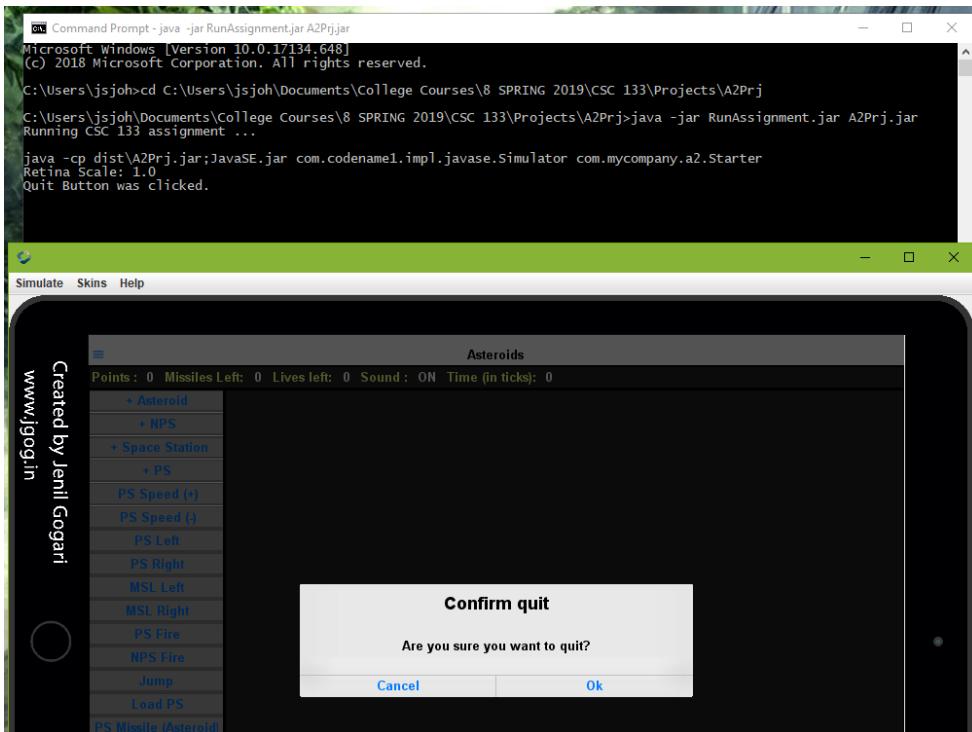


The screenshot shows the same Java application window as the previous one. The status bar now displays: Points: 0 Missiles Left: 10 Lives left: 3 Sound: ON Time (in ticks): 0. The button "+ Asteroid" is highlighted in blue, indicating it was recently clicked. The main game area remains visible.

Pressing “Q”:



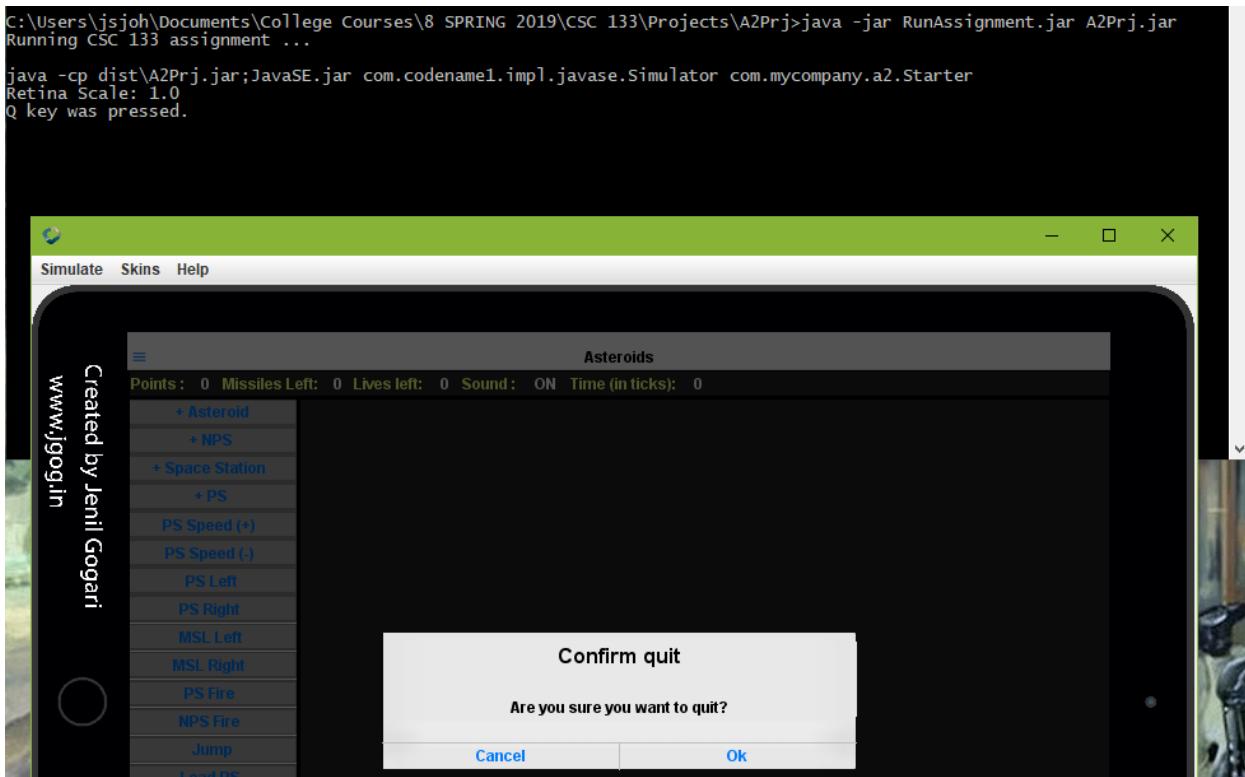
Quit Button from Side Menu:



Test Case 4: Closing the Program

Pressing Q:

```
C:\Users\jsjoh\Documents\College Courses\8 SPRING 2019\CSC 133\Projects\A2Prj>java -jar RunAssignment.jar A2Prj.jar
Running CSC 133 assignment ...
java -cp dist\A2Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a2.Starter
Retina Scale: 1.0
Q key was pressed.
```

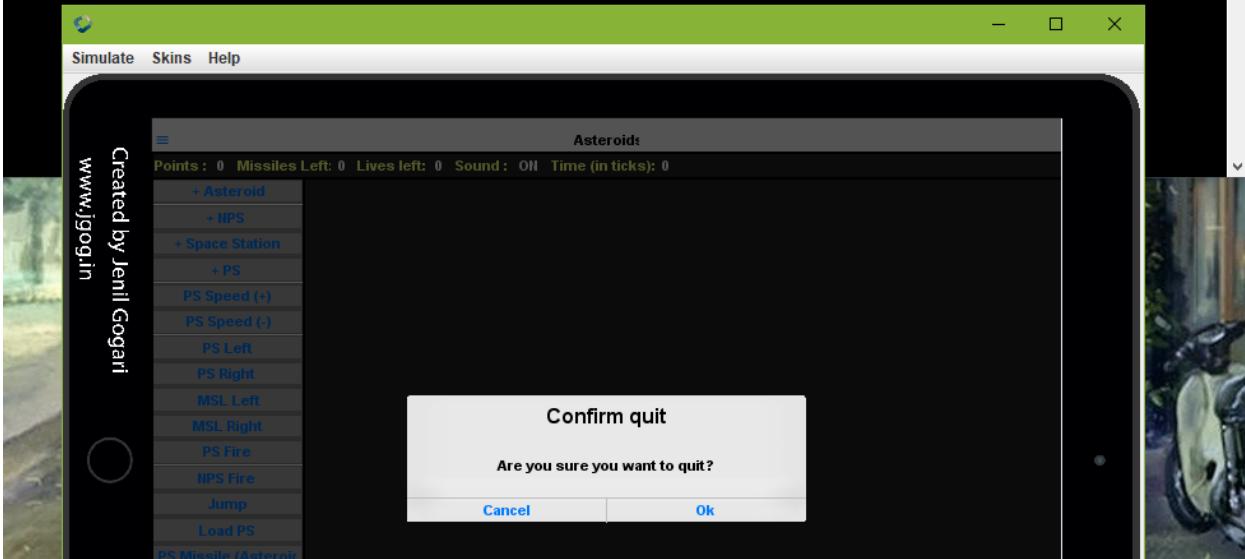


```
C:\Users\jsjoh\Documents\College Courses\8 SPRING 2019\CSC 133\Projects\A2Prj>java -jar RunAssignment.jar A2Prj.jar
Running CSC 133 assignment ...
java -cp dist\A2Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a2.Starter
Retina Scale: 1.0
Q key was pressed.

C:\Users\jsjoh\Documents\College Courses\8 SPRING 2019\CSC 133\Projects\A2Prj>
```

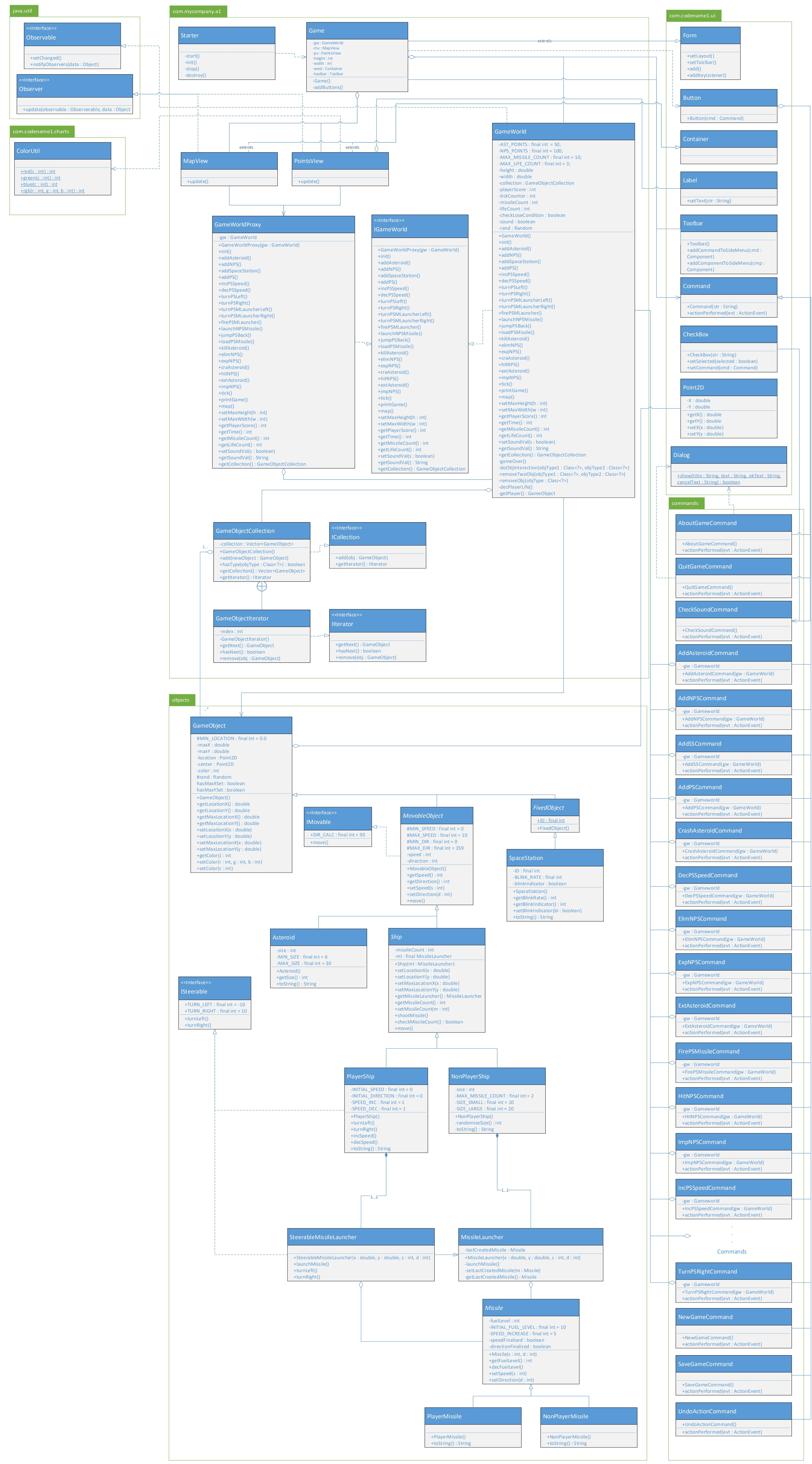
Quit Button:

```
C:\Users\jsjoh\Documents\College Courses\8 SPRING 2019\CSC 133\Projects\A2Prj>java -jar RunAssignment.jar A2Prj.jar
Running CSC 133 assignment ...
java -cp dist\A2Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a2.Starter
Retina Scale: 1.0
Quit Button was clicked.
```



```
C:\Users\jsjoh\Documents\College Courses\8 SPRING 2019\CSC 133\Projects\A2Prj>java -jar RunAssignment.jar A2Prj.jar
Running CSC 133 assignment ...
java -cp dist\A2Prj.jar;JavaSE.jar com.codename1.impl.javase.Simulator com.mycompany.a2.Starter
Retina Scale: 1.0
Quit Button was clicked.

C:\Users\jsjoh\Documents\College Courses\8 SPRING 2019\CSC 133\Projects\A2Prj>
```



Problem #3

Problem 3

John Pettet | April 2, 2019

