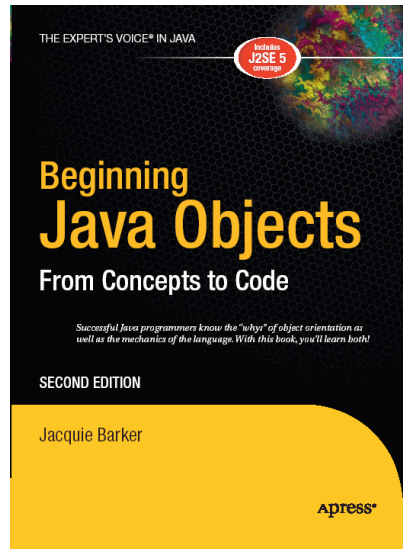


# Beginning Java Objects: From Concepts to Code

Jacque Barker



[jacque@objectstart.com](mailto:jacque@objectstart.com)

# Java and J2EE

## (C# and .NET)



- Java has gained widespread acceptance as a mainstream programming language
- Many organizations have standardized on Java 2 Enterprise Edition (J2EE) component technologies for building enterprise solutions
  - Servlets
  - JavaServer Pages (JSPs)
  - Enterprise Java Beans (EJBs)
- In order to use **J2EE** properly, one must be proficient with the basics of **Java**
- In order to be proficient with **Java** as an OOPL, one must in turn be "**object savvy**" ... herein lies the problem!

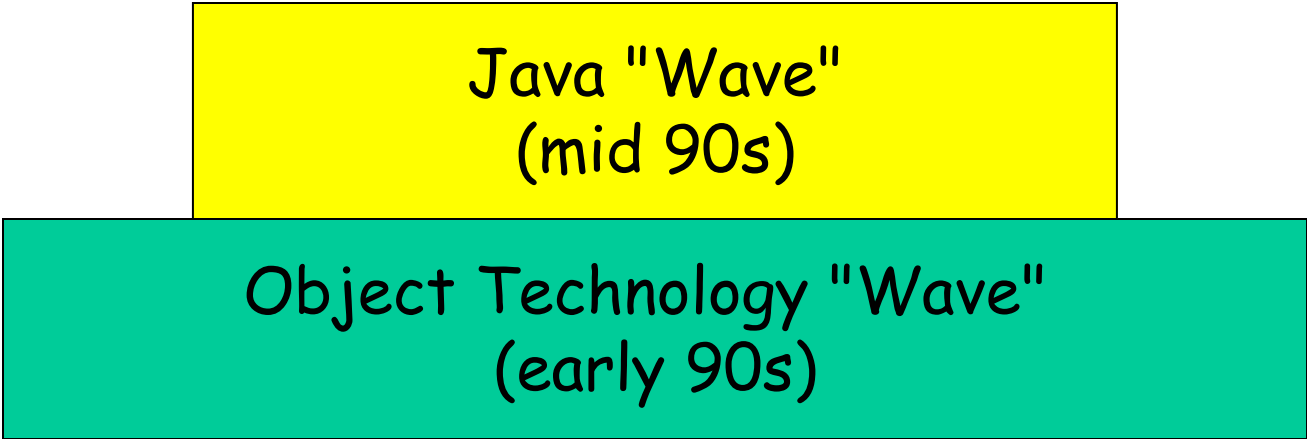
# A Fundamental Problem

Many procedural language programmers missed the initial object technology "wave" ...

Object Technology "Wave"  
(early 90s)

# A Fundamental Problem, cont.

... and the Java technology "wave" ...



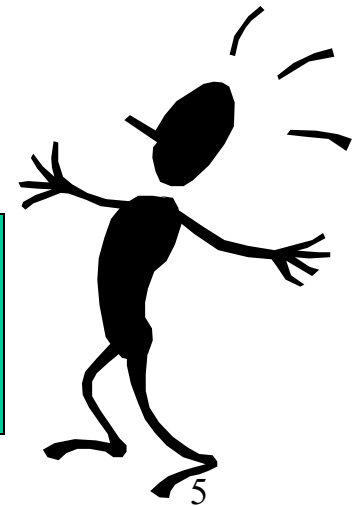
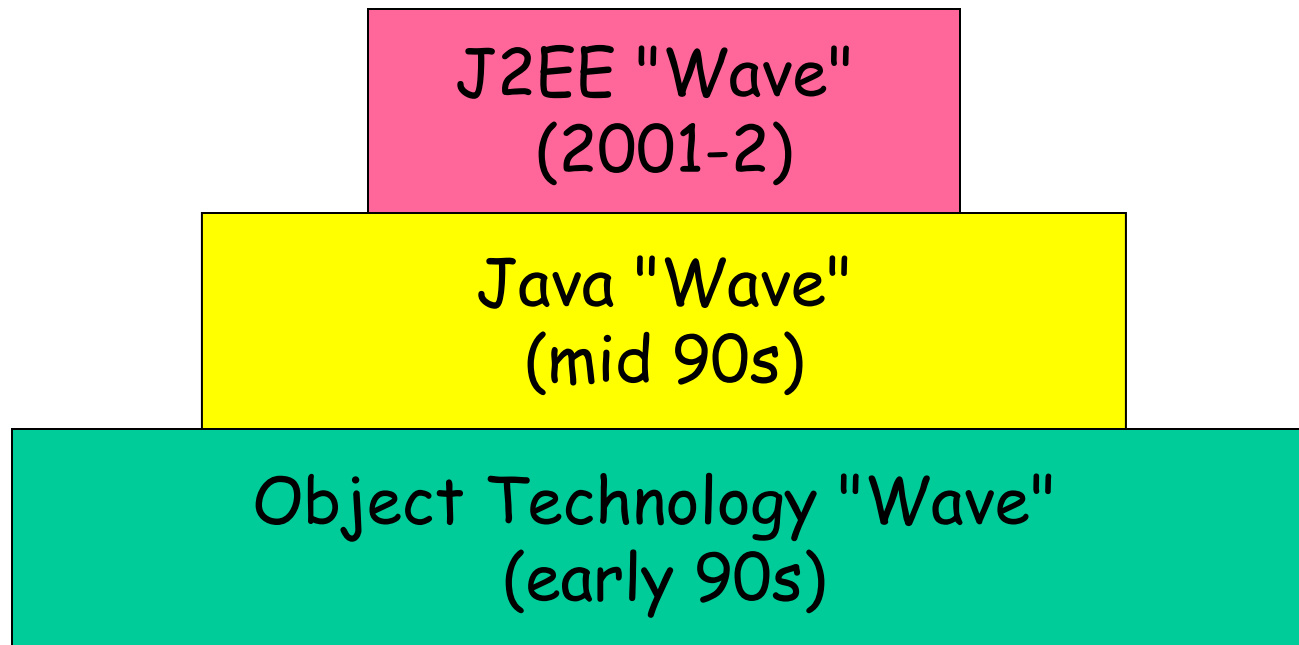
The diagram consists of two rectangular boxes. The top box is yellow and contains the text 'Java "Wave" (mid 90s)'. The bottom box is teal and contains the text 'Object Technology "Wave" (early 90s)'. The yellow box is positioned directly above the teal box, and its width is narrower than the teal box's width, centered horizontally relative to the teal box.

Java "Wave"  
(mid 90s)

Object Technology "Wave"  
(early 90s)

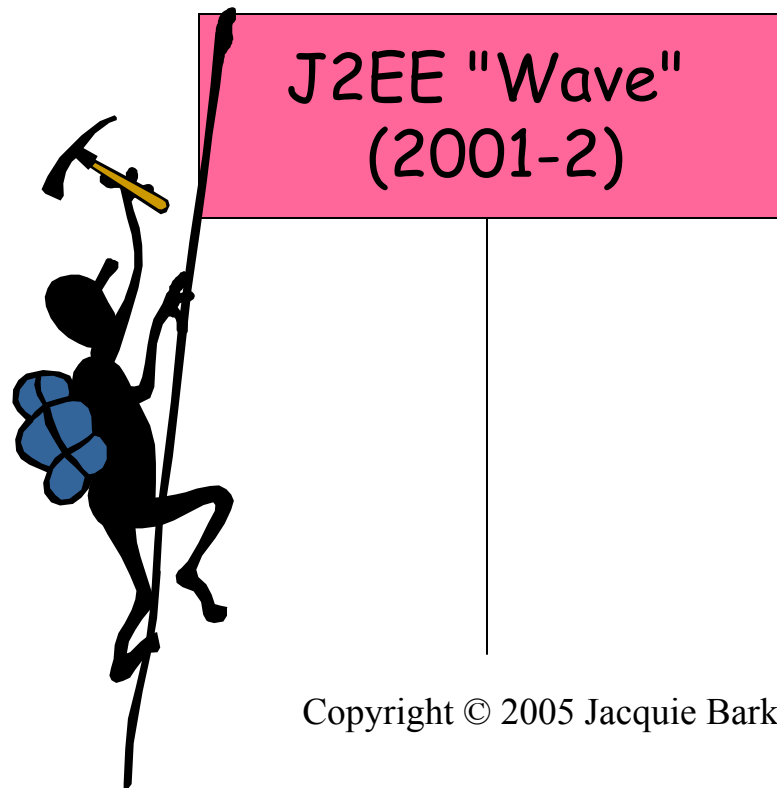
# A Fundamental Problem, cont.

... and are now faced with playing  
"catch-up" in a major way!



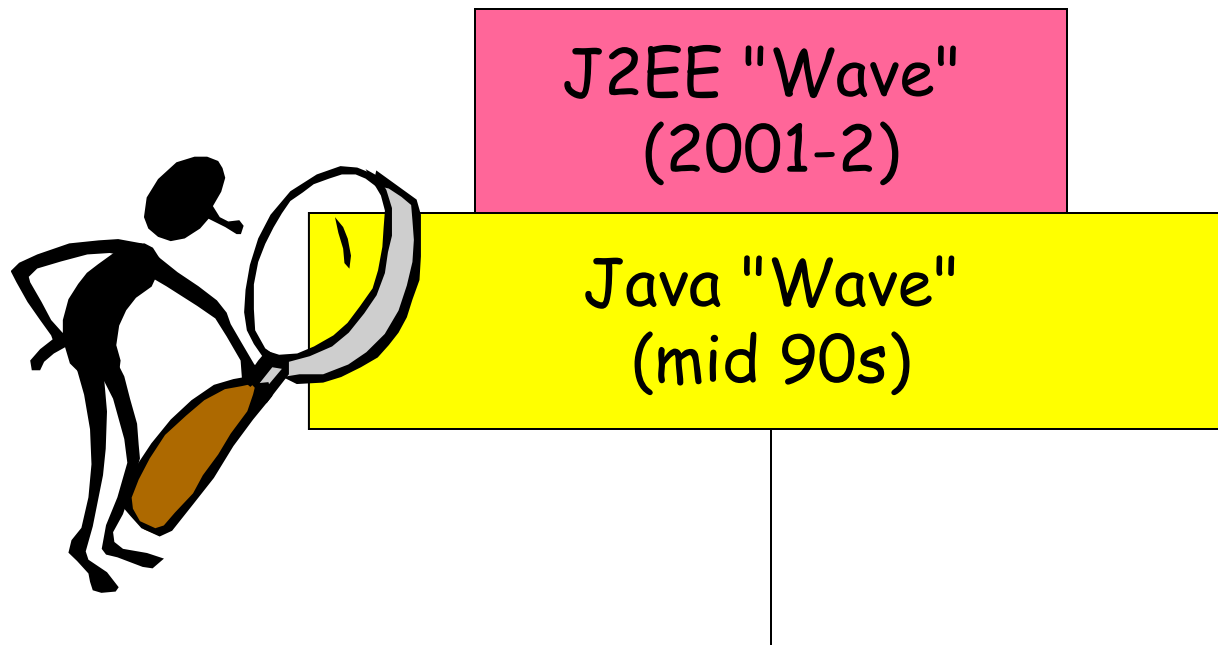
# A Fundamental Problem, cont.

Some are diving right in and trying to tackle J2EE component technologies directly ...



# A Fundamental Problem, cont.

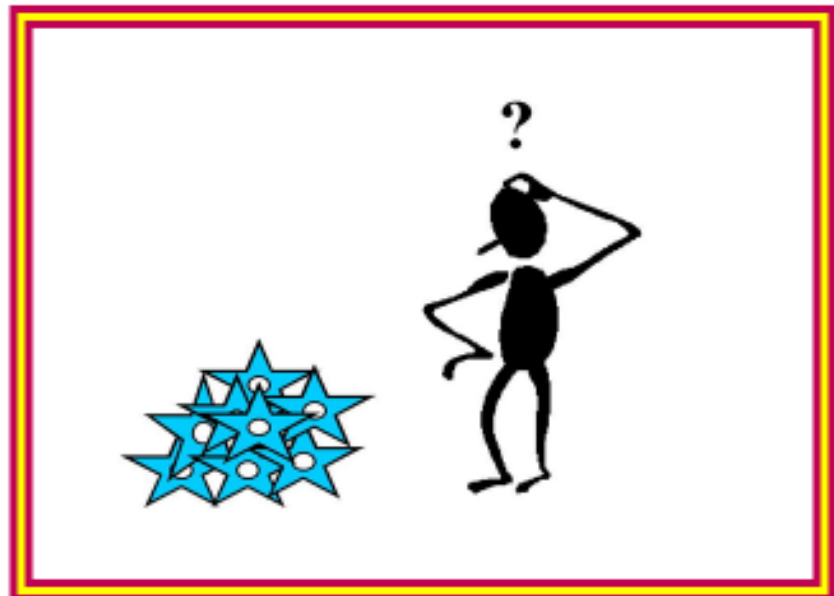
Others are stepping back to get some remedial Java training first ...



# A Fundamental Problem, cont.

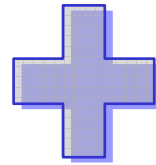
- Unfortunately, most approaches to teaching Java and/or J2EE **assume object proficiency** on the part of their audiences :op
- In order to get the most out of an object-oriented technology, one ***must*** be proficient with objects first!
  - A fable: The story of the famous homebuilder ...



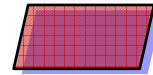




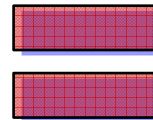
Strong general building skills



Powerful new building materials



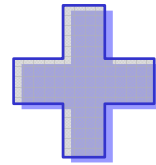
Lack of knowledge of their properties



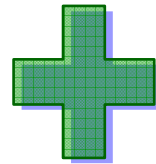
Inflexible, difficult to maintain result ...



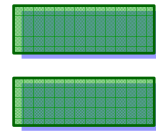
Strong general building skills



Powerful new building materials



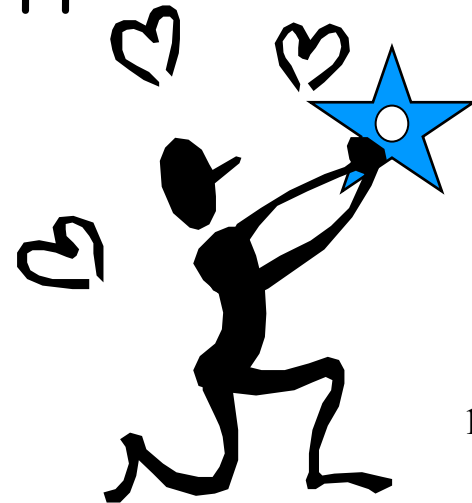
Full appreciation for their properties



Dramatically improved result!!!

# The "Object Crisis" (TM)

- Software developers are faced with a similar challenge
  - Often thrown onto a J2EE project without the proper Java -- and **object!** -- foundation
  - The result: a tangled mess when it comes to **maintainability** over the application's lifecycle
- If properly architected, J2EE applications should EMBRACE objects!



# The "Object Crisis ", cont.

- We need to focus on how best to "retool" experienced software engineers in the **object paradigm**, then in **Java**, then in **J2EE** (or in objects, then C#, then .NET)
- This course is a perfect first step!

# ObjectStart Training

- Beginning Java Objects (4 days)
  - Lecture/lab format
  - Teaches fundamental object oriented programming concepts side by side with Java language syntax
  - This course is an ideal vehicle for “retooling” a procedural programming team with objects in general and with Java in particular
  - No prior experience with either objects or Java is expected on the part of the participants

# ObjectStart Training, cont.

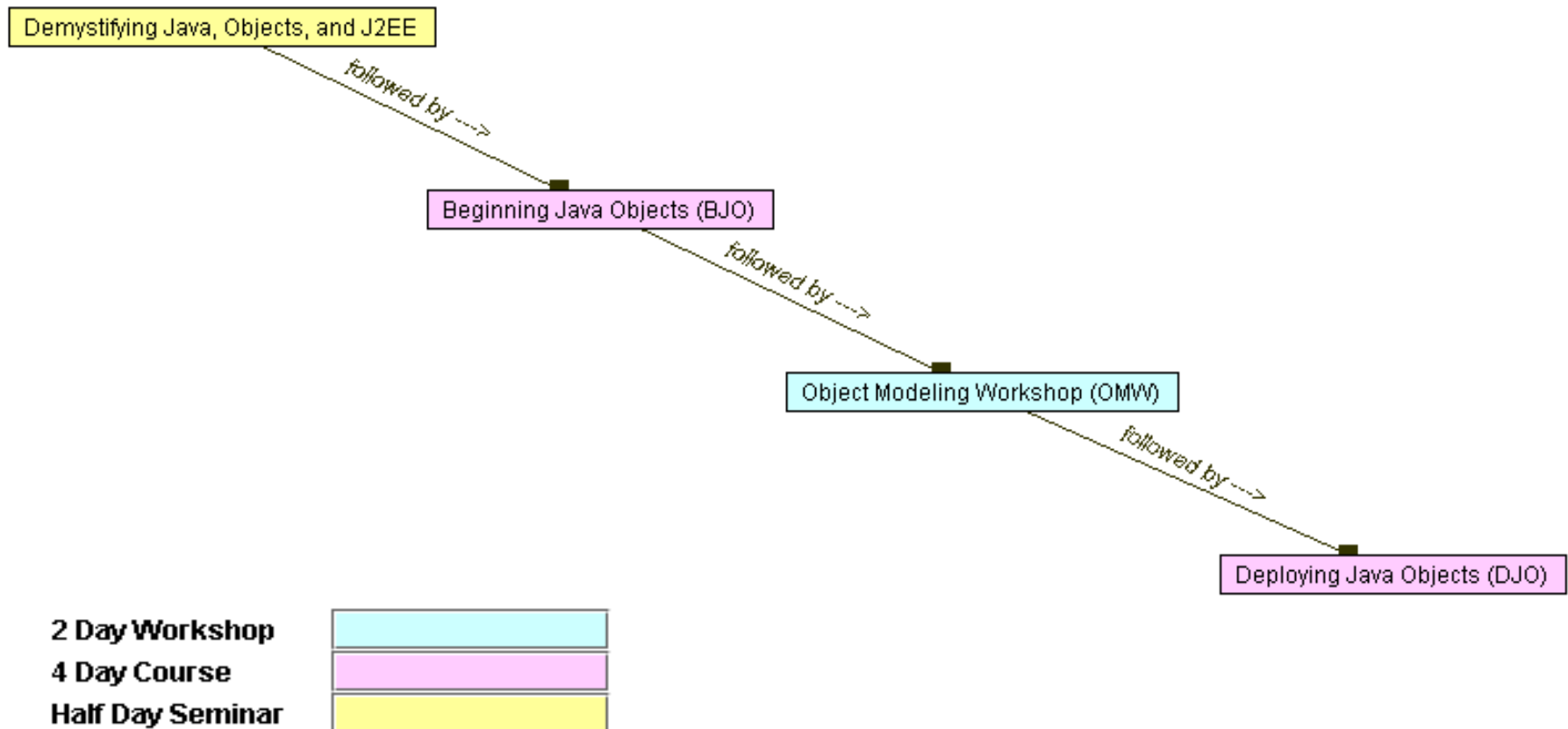
- Object Modeling Workshop (1 day)
  - Informal workshop designed to serve as an add-on to the BJO course
  - Goal: once students have learned the fundamentals of objects and Java, give them immediate hands-on experience with crafting an object model "blueprint"
  - Even folks who have previously attended object modeling/UML training will benefit from this workshop

# ObjectStart Training, cont.

- Deploying Java Objects (4 days)
  - Lecture/lab format
  - How to use core J2EE component technologies (servlets, JSPs, and JDBC) to build thin-client web applications
  - How to use the Java AWT and Swing APIs to build classic desktop client-server applications
  - Emphasis on key architectural principles of **model-data layer separation** and **model-view separation**



# Recommended Learning Track



# Intended Audience for the Course

- Anyone who has yet to tackle Java (or, any OOP), but who wants to get off on the right foot with the language
- Prerequisite: general programming proficiency
  - Simple data types (integer, floating point, etc.)
  - Variables and their scope (including the notion of global data)
  - Control flow (if-then-else statements, for/do/while loops, etc.)
  - What arrays are, and how to use them

# Intended Audience, cont.

- The notion of a function/subroutine/subprogram: how to pass data in and get results back out
- Using the language of your choice, can you easily write a program which uses looping constructs to print the following output?

```
1 :  *
2 :  **
3 :  ***
4 :  ****
5 :  *****
```

# Intended Audience, cont.

- Also appropriate for:
  - People who have been exposed to Java syntax, but who don't know how to structure an application to best take advantage of Java's OO features
  - Anyone who has used a Java IDE, but only knows how to drag and drop GUI components without a real sense of how to structure the core of the application around objects

# Organization of the Book

- Part 1: The ABC's of Objects (*"bricks and mortar"*)
  - Start with simple concepts -- What is an object? What is a class?
  - Graduate to rather advanced object concepts such as polymorphism, abstract classes, and interfaces
  - Provide a gradual introduction to Java syntax by illustrating how each concept is implemented in code

**\*\*\* This is the primary emphasis  
for this course \*\*\***

# Organization, cont.

- Objects are for the most part 'language neutral'
- What you'll learn conceptually about Java objects in this course applies equally well to:
  - C++
  - Ada
  - Eiffel
  - Smalltalk
  - An as-yet-to-be-invented OO languageunless otherwise noted

# Organization, cont.

- Part 2: Object Modeling 101 (*"architect"*)
  - Introduce the generic process of object modeling, and the basics of UML notation
  - Somewhat irreverent!
  - Based on a Student Registration System case study (see page 8 in the book)
- In this course, we'll only have time for a brief review of UML notational syntax

# Organization, cont.

- Part 3: Translating an Object "Blueprint" into Java Code (*"home builder"*)
  - Illustrate how to translate the UML model for the SRS into a fully-functional Java application with:
    - File persistence
    - GUI front-end
- We'll be covering the first step in this process in this course: building a command-line driven application
  - The relevance of doing so will become clear by the end of the course



# Organization, cont.

- We won't be covering the following in this course:
  - Graphical user interface (GUI) development
  - Distributed/Web deployment
  - The Java DataBase Connectivity (JDBC) API
  - Java 2 Enterprise Edition (J2EE) technologies
    - Servlets
    - Java Server Pages (JSPs)
    - Enterprise Java Beans (EJBs)

# Target Schedule

Day 1:	Chapters 1, 2, 3, 4 (slides 1 – 223)	Exercises #1, #2, #3, #4
Day 2:	Review; Chapter 5 (slides 224 – 349)	Exercises #5, #6, #7
Day 3:	Review; Chapters 6, 7 (slides 350 – 502)	Exercises #8, #9, #10, #11
Day 4:	Review; Chapters 10, 13; J2EE Concepts (slides 503 – end)	

# We'll Remain Flexible!

- Try to have exercises fall at lunch breaks or at the end of the day
  - Lunch break will be one hour long, and will occur some time between 11:30 and 1 PM
- We'll adjust the pace as appropriate
  - If we run a bit behind, I will skip a few of the less essential details
  - If we get ahead, I have plenty of extra material that I can cover at the end of the last day

# A Little Bit About Me ...

- Teaching objects/OOPL for 9 years
  - Adjunct faculty member, GWU's Graduate School of Business
  - Corporate training
- Hands-on software engineer
  - Programming in Java for 7 years, 27 years total
- Author
  - Beginning Java Objects, Second Edition (ISBN 1590594576)
  - Beginning C# Objects (ISBN 159059360X)
  - Taming the Technology Tidal Wave (ISBN 0974479888)

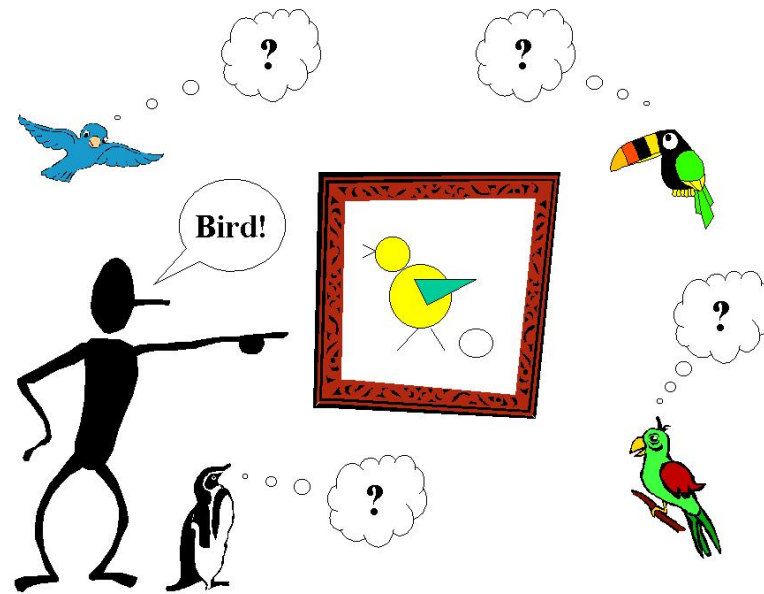
# Getting to Know You

How many of you ...

- ... meet the prerequisites for this course?
- ... are hands-on software developers? In which languages?
- ... are aspiring to BECOME software developers?
- ... are managers who want to better UNDERSTAND software developers? (good luck ;o)
- ... have "dabbled" in an OOPL?
- ... have "dabbled" with Java specifically?
- ... have previously studied object concepts?

# Logistics/ Questions?

# Abstraction as a Basis for Software Development (Chapter 1)



# Definition

- **Abstraction:** the process of
  - (a) recognizing, focusing on, and organizing the important characteristics of a situation or object, and
  - (b) filtering out/ignoring all of the unessential, distracting details
    - Example: describe this classroom ...

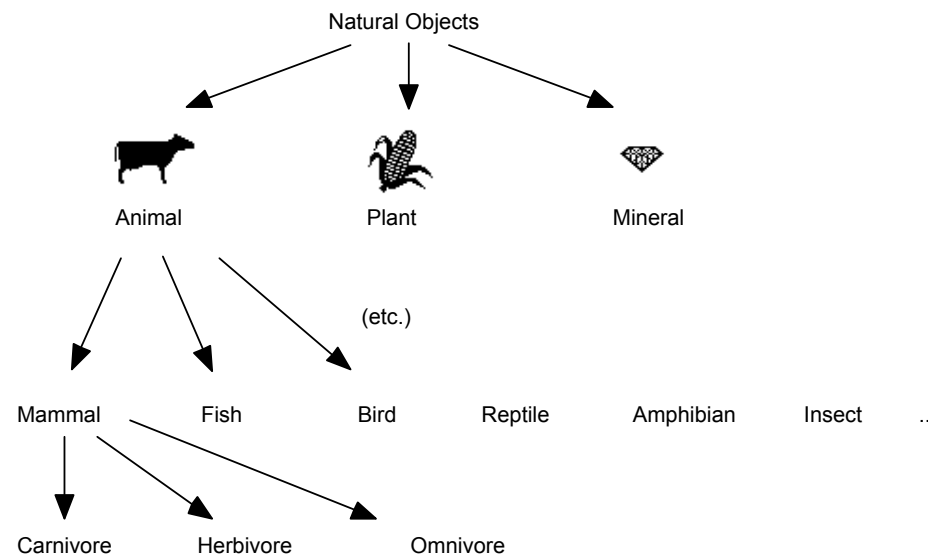


# Classification Hierarchies

- Even though our brains are adept at abstraction, that still leaves us with hundreds of thousands, if not **millions**, of separate abstractions to deal with over our lifetimes
- To cope, human beings systematically arrange information into categories according to established criteria; this process is known as **classification**

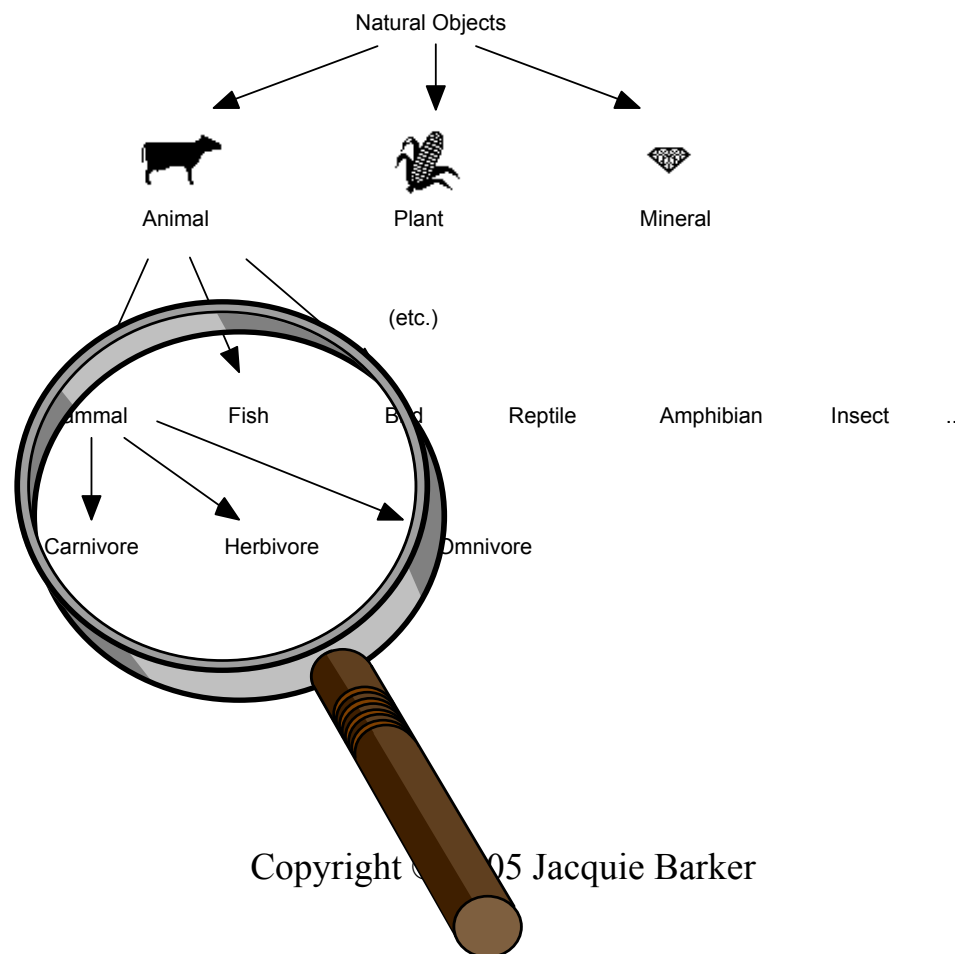
# Classification Hier., cont.

- We naturally organize concepts into hierarchies ...



# Classification Hier., cont.

- ... and then we mentally step up and down the hierarchy, focussing on only a single layer or subtree

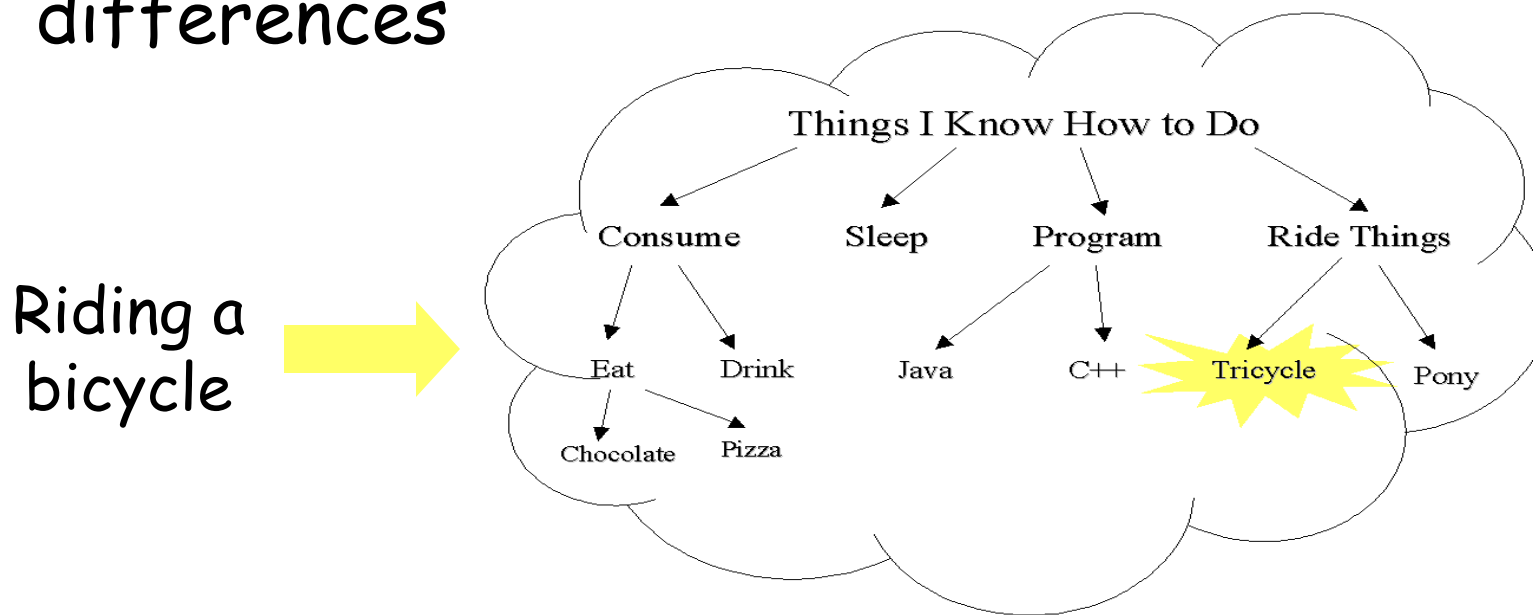


## Classification Hier., cont.

- By doing so, we reduce the number of concepts that we mentally need to 'juggle' at any one time
- No matter how complex a classification hierarchy grows to be, it needn't overwhelm us if it is properly organized

# Searching the Hierarchy

- When we are learning a new concept, we search our mental hierarchy for as similar a concept as we can find, so as to focus only on the differences



# Abstraction and Software

- When pinning down the requirements for a software application, we must make a judgement call as to which details of the “real world” are relevant
  - To err on the side of including too little detail will produce a model that doesn't fulfill its mission
  - To err on the side of including too much detail will overly complicate the resultant model, making it that much more difficult to design, program, test, debug, and extend in the future

# Abstraction & Software, cont.

- As with all abstractions, our decisions of inclusion vs. elimination must be made within the context of the overall **domain** of the future application
  - When representing a person, is his/her eye color important? How about his/her genetic profile? Salary? Hobbies?

# Abstraction & Software, cont.

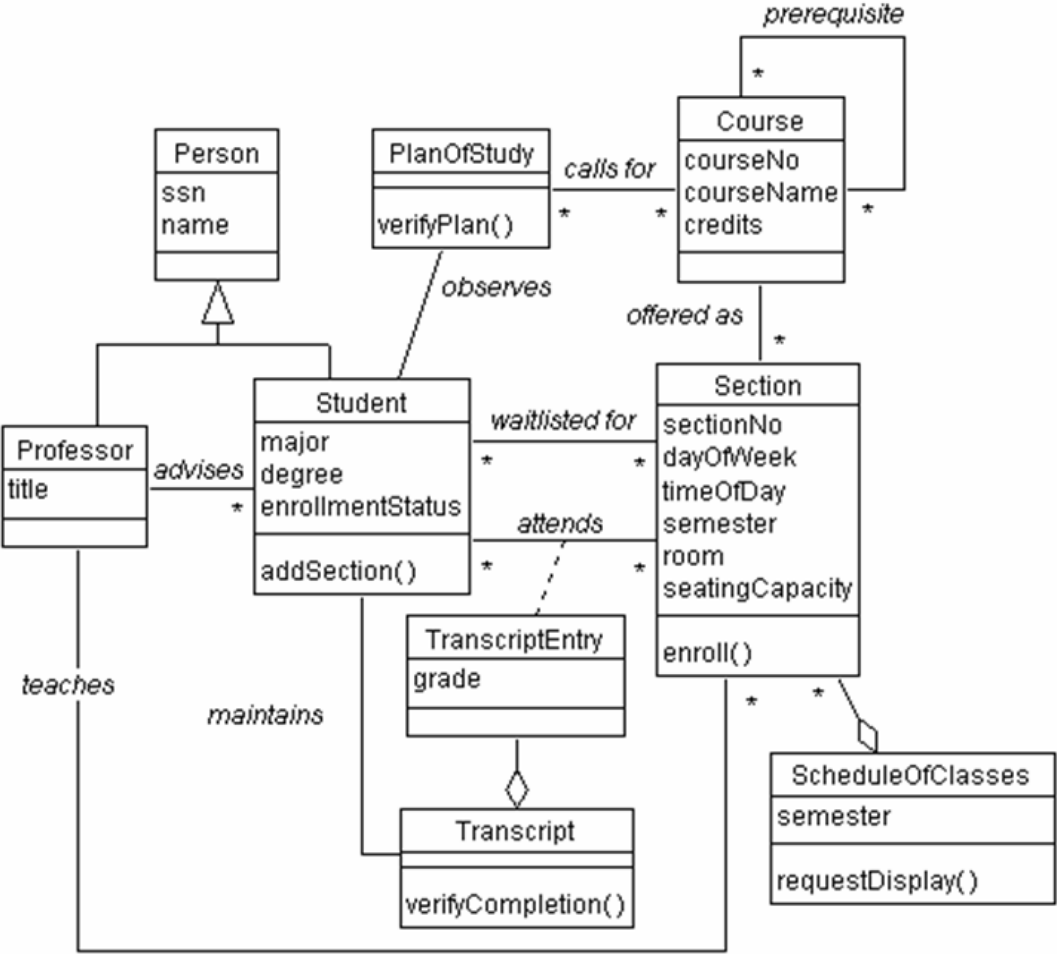
- As with all abstractions, our decisions of inclusion vs. elimination must be made within the context of the overall **domain** of the future model
  - When representing a person, is his/her eye color important? How about his/her genetic profile? Salary? Hobbies?
  - Any of these features may be relevant or irrelevant, depending on whether the domain of the model to be developed involves:
    - » Marketing demographics
    - » Economic trends
    - » Terrorist networks



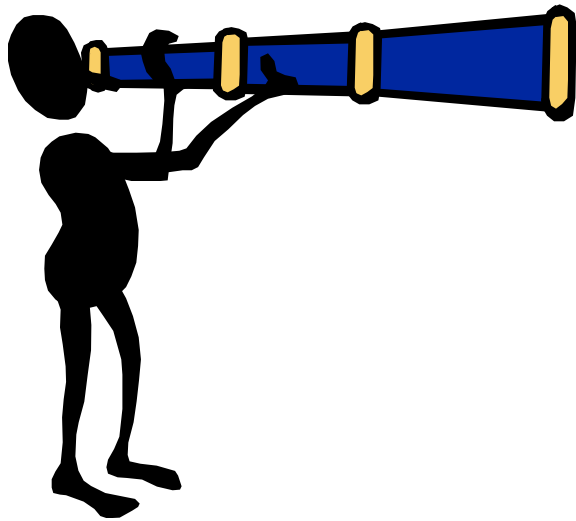
# Abstraction & Software, cont.

- Once we've determined the essential aspects of a situation, we can prepare a **model**
  - A "blueprint" for building the software
- We'll use the Unified Modeling Language (UML) to render such blueprints
- By the end of the semester, UML diagrams such as that shown on the next slide for the SRS should make perfect sense

# SRS (UML)



# A First Glimpse of the Java Language (Chapter 2)



Java

# Gradual Introduction

- Chicken vs. egg: should learn object concepts first, but it is best to illustrate object concepts with a OOPL
- We'll use pseudocode to skip premature details

```
for (int i = 0; i <= 10; i++) {    ← Real Java syntax.  
    compute the grade for the ith Student ← Pseudocode!  
}
```

# Anatomy of a Simple Java Program

```
// Simple.java  
//  
// A trivially simple example for illustrating the anatomy  
// of a (non-OO) Java program.  
//  
// Written by Jacquie Barker.
```

} introductory  
comment

class  
"wrapper" {

```
public class Simple {  
    public static void main(String[] args) {  
        System.out.println("Whooo!!!!!!");  
    }  
}
```

} main  
method

# Anatomy, cont.

- Comment syntax

- The C language style of block comments:

```
/* This is a single line C-style comment. */
```

```
/*  
x = y + z;  
a = b / c;  
j = s + c + f;  
*/
```

- The C++ single line form of comment:

```
x = y + z;    // Text of comment starts here through the end of the line.
```

```
// Here is a block of C++ style comments.  
// This serves as an alternative to using the C style  
// of block comments (/* ... */).  
m = n * p;
```

# Anatomy, cont.

- "Javadoc" comments, which enable you to automatically generate HTML documentation for your application (see <http://java.sun.com/j2se/javadoc/> )

```
/** This method is used to double the value of an integer.  
 *  
 * @param x - the value to be doubled.  
 */  
public int doubleIt(int x) {  
    return 2 * x;  
}
```

Using a special command-line utility, produces HTML documentation:

## Method Detail

### **doubleIt**

```
public int doubleIt(int x)
```

This method is used to double the value of an integer.

#### **Parameters:**

x - the value to be doubled.

# Anatomy, cont.

- The main function/**method**
  - Drives the program
  - Must have the following "signature":

```
public static void main(String[] args)
```
  - We'll discuss the significance of this signature as we proceed



# Anatomy, cont.

- The main method lives in a class “wrapper”

```
public class ClassName {  
    public static void main(String[] args) {  
        code of main method goes here ...  
    }  
}
```

- E.g.,

```
public class Simple {  
    public static void main(String[] args) {  
        System.out.println("Whee!!!!!!");  
    }  
}
```

- The class name starts with a capital letter; can be any non-reserved name
- The class is typically declared **public** (we'll learn the significance of this later on)

# Anatomy, cont.

- Place a "public" class in a file by the same name, ending with ".java": `Simple.java`
  - Pay close attention to use of upper/lower case -- it must match precisely!
- To compile: `javac Simple.java`
  - Yields a file named `Simple.class`
- To run: `java Simple`

# Case Sensitivity

- Even though Windows/DOS is not case sensitive, we must pay careful attention to case when dealing with Java
  - Externally, when naming files
    - If the Simple class is saved in a file named "simple.java", for example, the compiler will complain:

class Simple is public, should be declared in a  
file named Simple.java

# Case Sensitivity, cont.

- Internally to our programs
  - Variables "x" and "X" are different!
  - Reserved words must be used exactly as presented in this course

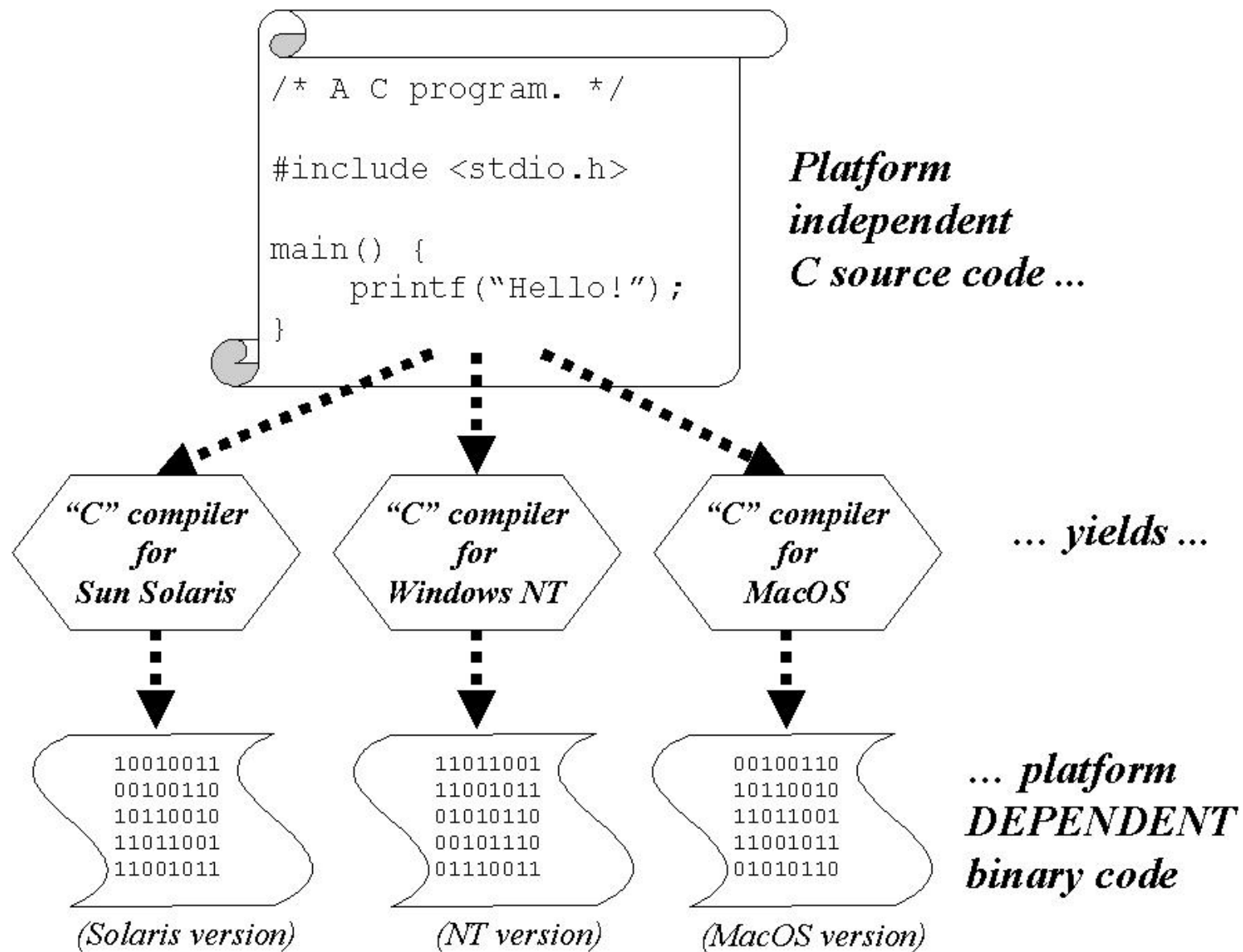
# Exercise #1

# Mechanism of Java

- Let's look "behind the scenes" at the mechanism of Java as an interpreted language
- We'll start by discussing a conventionally compiled language, then contrast Java's approach

# Conventional Compilation

- With a compilable language like C or C++, source code must first be compiled into an executable form known as **binary code** or **machine code**
  - a pattern of 1's and 0's understandable by the underlying **hardware architecture** of the computer on which the program is intended to run
- The resultant executable version will be tied to a particular platform's architecture, and is executed by the operating system





# Architecture Neutral

- In contrast, Java source code is not compiled into binary, but rather into a special format known as **bytecode**
- Bytecode is **architecture neutral**
  - No matter whether a Java program is compiled under Windows XP, or Linux, or any other operating system for which a Java compiler is available ...
  - ... the resultant bytecode turns out to be the same!

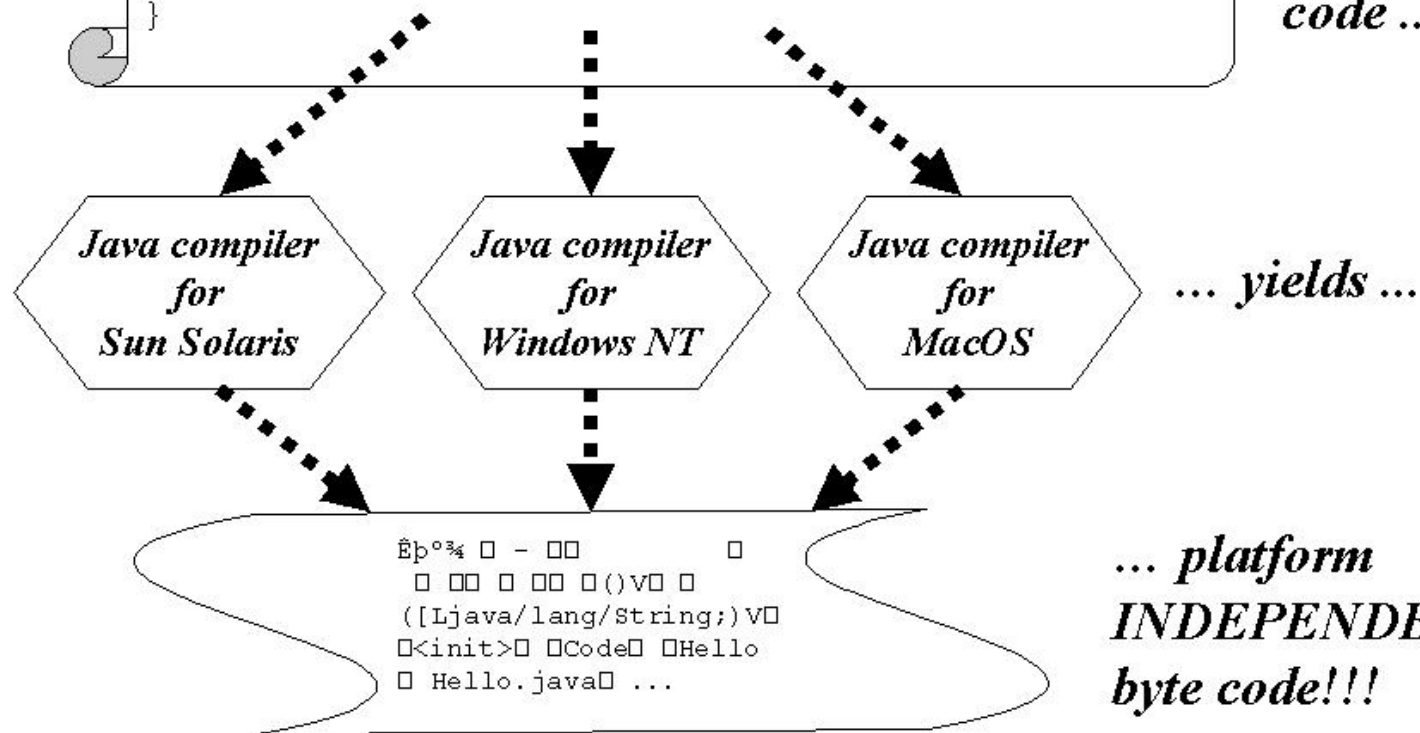
```

/* A Java program. */

public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello!");
    }
}

```

*Platform  
independent  
Java source  
code ...*

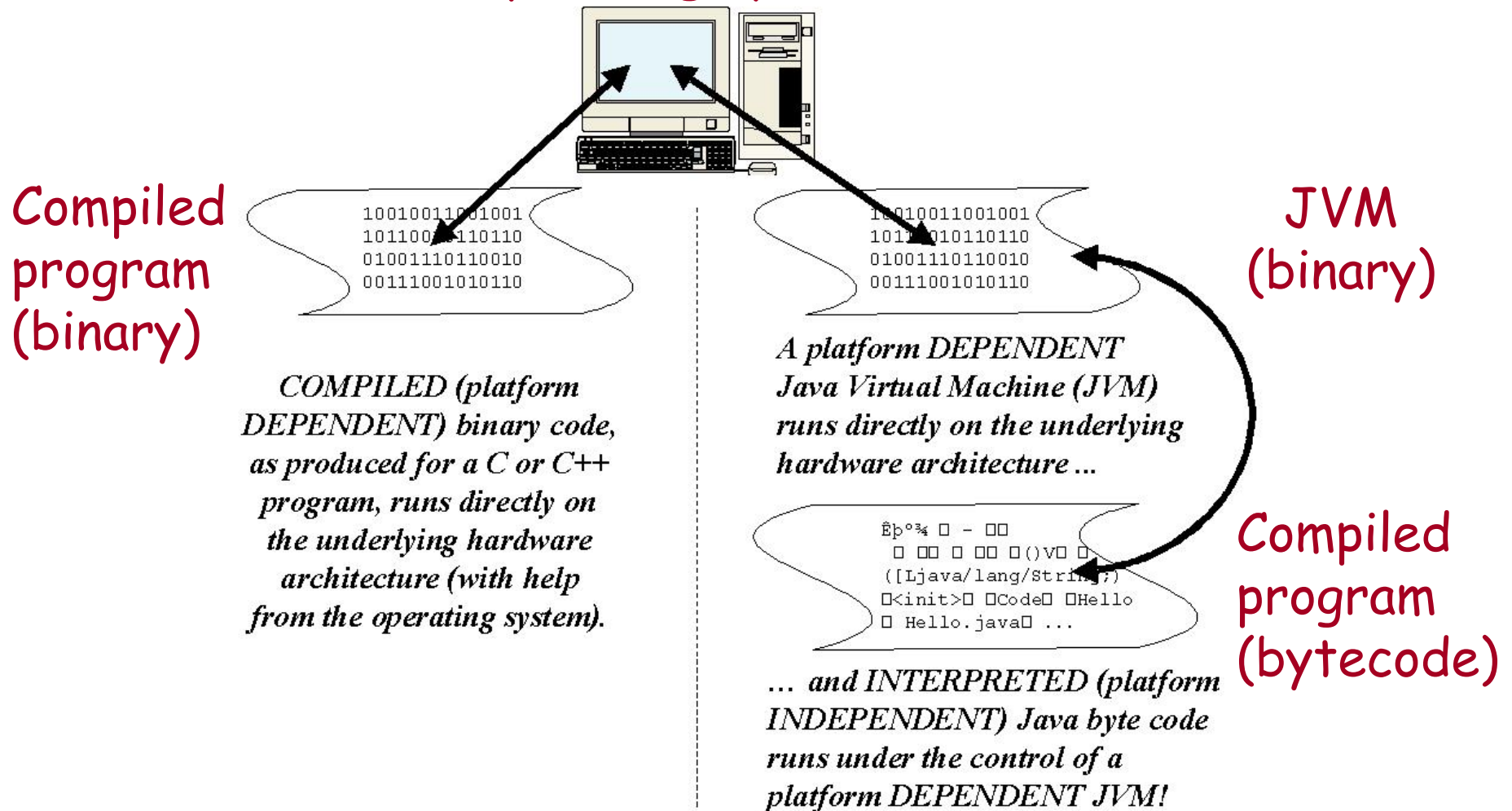


# Architecture Neutral, cont.

- Bytecode is not executed directly by the operating system the way that binary/machine code is
- To execute bytecode, we need a (platform-dependent) **Java Virtual Machine (JVM)** for the target machine
  - Analogy: CD player

- We execute byte code with a *platform dependent JVM*

(Operating system)



# Architecture Neutral, cont.

- The architecture neutrality of bytecode is precisely why Java became so popular at the time that the World Wide Web was “exploding”
- Java was used to write **applets**, downloadable “mini programs” that are referenced in an HTML page
  - <APPLET CODE=MyApplet WIDTH=80 HEIGHT=100></APPLET>
  - Downloaded to your local (client) computer
  - Run under control of your browser
- A way of providing dynamic content to the web

# Architecture Neutral, cont.

- Java has come a long way since the advent of applets
- Server side Java components are now also platform independent
  - Java 2 Enterprise Edition (J2EE)
    - » Servlets
    - » Java Server Pages (JSPs)
    - » Enterprise Java Beans (EJBs)

# Mechanics, cont.

- When we type

```
java Simple
```

therefore, we are starting up the JVM, and telling it to:

- Look for a bytecode file called `Simple.class`
- Load the bytecode into the JVM's memory
- Execute its `public static void main(String[] args)` method

# Looking for Bytecode

- The JVM will by default look in
  - The current working directory for user-created classes, and
  - The Java "home" directory for built-in classes
- Can set up a `CLASSPATH` environment variable to point to other directories/Java archive (.jar) files (e.g., `src.jar`)



# JVM Errors

- If there is no such class file, or the class file doesn't contain the appropriate main() method signature, we'll get an error from the JVM

- Class file missing entirely:

```
java Foo
```

```
Exception in thread "main"
```

```
java.lang.NoClassDefFoundError: Foo
```

- Missing main() method:

```
java Foo
```

```
Exception in thread "main"
```

```
java.lang.NoSuchMethodError: main
```

# Why Java?

- Platform independent, aka architecture neutral
- Java's elegance as an OO language
- Vendor neutral
- Open standard/open source, and hence low cost of entry

# Simple/Built-In Data Types

- Numerics:
  - byte: 8 bit integer (range -128 to 127)
  - short: 16 bit integer (range -32,768 to 32,767)
  - int: 32 bit integer (range  $-2^{31}$  to  $2^{31} - 1$ )
  - long: 64 bit integer (range  $-2^{63}$  to  $2^{63} - 1$ )
  - float: 32 bit floating point number
  - double: 64 bit floating point number
- char - a single 16 bit Unicode character  
`char c = 'A'; // Use single quotes to enclose`

# Built-In Data Types, cont.

- **boolean:** may only take on the reserved values **true** **and** **false**

```
boolean error = false; // Initialize a flag.  
// ...  
// Later in the program:  
if (some situation arises) error = true;  
// ...  
// Still later in the program:  
if (error) take some corrective action;
```

# Built-In Data Types, cont.

- `String`: a series of characters
  - Note capital "S" -- this is significant, for reasons we'll discuss later

```
String aString = "I am a string!"; // note double quotes
```

- The plus sign (+) operator is used to concatenate Strings:

```
String x = "foo";
```

```
String y = "bar";
```

```
String z = x + y + "!"; // z now equals "foobar!"
```

# Initializing Variables

- Generally speaking, when variables of any of these data types are declared, their values are not automatically initialized
- Trying to access variables without explicitly initializing them will result in a compilation error

```
public static void main(String[] args) {  
    int i;    // not automatically initialized  
    int j;    // ditto  
    j = i;    // compilation error!  
}
```

**"Variable i may not have been initialized."**

# Initializing Variables, cont.

- It is a good practice to explicitly initialize all variables, primarily as a communication tool:

```
public static void main(String[] args) {  
    // Explicit initialization.  
    int i = 0;  
    char c = ' '; // a blank space  
    double d = 0.0;  
    String s = ""; // an "empty" String  
    // etc.
```

# Initializing Variables, cont.

- The rules of automatic initialization are different when dealing with the 'inner workings' of objects, which we'll talk about later



# Expressions

- **A simple expression** in Java is either:
    - A constant: 7, false
    - A String literal: "foo"
    - A char(acter) literal: 'A'
    - A variable declared to be of one of the built-in types that we've seen so far: aString, x
    - Any two expressions combined with one of the Java **binary operators**: `x + 2`
    - Any expression that is modified by one of the Java **unary operators**: `i++`
    - Any of the above, enclosed in parentheses: `(x + 2)`
- plus a few more types related to objects

# Expressions, cont.

- Expressions of arbitrary complexity can be built up around these different simple expression types by nesting parentheses - e.g.,

$$(x + ((y/z) + z) * 7) + y$$

- We evaluate such expressions from innermost to outermost parentheses, left to right

# Expressions, cont.

- Assume  $x = 3, y = 4, z = 1$ :

$$(x + ((y/z) + z) * 7) + y$$

# Expressions, cont.

- Assume  $x = 3, y = 4, z = 1$ :

$(x + ((y/z) + z) * 7) + y$

$(x + ((4 + z) * 7) + y)$

# Expressions, cont.

- Assume  $x = 3, y = 4, z = 1$ :

$(x + ((y/z) + z) * 7) + y$

$(x + (4 + z) * 7) + y$

$(x + (5 * 7) + y)$

# Expressions, cont.

- Assume  $x = 3, y = 4, z = 1$ :

$(x + ((y/z) + z) * 7) + y$

$(x + ((4 + z) * 7) + y)$

$(x + (5 * 7) + y)$

$(x + 35 + y)$

# Expressions, cont.

- Assume  $x = 3, y = 4, z = 1$ :

$(x + ((y/z) + z) * 7) + y$

$(x + ((4 + z) * 7) + y)$

$(x + (5 * 7) + y)$

$(x + 35 + y)$

$(38 + y)$

# Expressions, cont.

- Assume  $x = 3$ ,  $y = 4$ ,  $z = 1$ :

$$(x + ((y/z) + z) * 7) + y$$
$$(x + ((4 + z) * 7) + y)$$
$$(x + (5 * 7) + y)$$
$$(x + 35 + y)$$
$$(38 + y)$$
$$42$$



# Autoincrement/decrement

- Java provides **autoincrement** (++) and **autodecrement** (--) operators: `i++;`
  - Increase or decrease the value of a variable by:
    - 1, for bytes, shorts, ints, and longs;
    - 1.0, for floats and doubles;
    - One alphabetic character, for chars (e.g., 'A' becomes 'B')

# Autoincrement/decrement, cont.

- Prefix (++i) vs. postfix (i++) application:
  - If the operator comes at the end of a variable name ("after"), the variable's value is altered after its value is used in evaluating an expression
  - Prefix: if the operator comes before, the variable's value is altered before the expression to which it belongs is evaluated

# Autoincr./decr., cont.

```
int i, j, k;
i = 1;
j = i++;    // j receives the value 1, because i is
            // incremented from 1 to 2 AFTER its value
            // is used in the assignment statement.
            // Equivalent to:
            //     j = i;
            //     i = i + 1;

k = ++i;    // k receives the value 3, because i is
            // incremented from 2 to 3 BEFORE its
            // value is used in the assignment statement.
            // Equivalent to:
            //     i = i + 1;
            //     k = i;

// Can also stand alone as a statement:
i++; // equivalent to: i = i + 1;
```

# Automatic Type Conversion and Explicit Casting

- Java supports automatic type conversions
  - If you try to transfer the value of some variable *y* to another variable *x*, of a different type:

```
double x;  
int y;  
y = 3;  
x = y; // int 3 is converted to double 3.0
```

then Java will attempt to automatically convert the type of *y* to match the type of *x*, if no precision is lost in doing so

```
int x;  
double y;  
y = 2.7;  
x = y; // Compiles in C and C++, but not in Java.
```

# Automatic Type Conversion and Explicit Casting, cont.

- **Error:** Incompatible type for =. Explicit cast needed to convert double to int.
- In order to signal to the Java compiler that we are willing to accept the loss of precision, we must perform an **explicit cast**

```
int x;  
double y;  
y = 2.7;    // 2.7F if y was declared to be float ...  
x = (int) y; // This will compile in Java now!
```
- We'll see other applications of casting, involving objects, later

# Printing to the Screen

- To print text messages to the command line window:

```
System.out.println(String expression);
```

- Don't worry about the 'strange' syntax

- The `System.out.println()` method can accept very complex expressions, and does its best to ultimately turn these into a single `String` value

```
System.out.println("Hi!");  
String s = "Hi!";  
System.out.println(s);  
String t = "foo";  
String u = "bar";  
System.out.println(t + u); // prints "foobar".
```

# Printing to the Screen, cont.

```
int x = 3;
int y = 4;

// Prints "3".
System.out.println(x);

// Prints "7". So far, so good!
System.out.println(x + y);

// Prints "7 was the answer".
System.out.println(x + y + " was the answer");

// Whoops! Prints "The sum is: 34".
System.out.println("The sum is: " + x + y);

// Correctly prints "The sum is: 7".
System.out.println("The sum is: " + (x + y));
```

# Printing to the Screen, cont.

- The statement:

```
System.out.print(the String expression to be printed);
```

omits the hard return at the end of each line

```
System.out.print("J");  
System.out.print("AV");  
System.out.println("A");    // prints "JAVA"
```

- Printing a blank line by itself:

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



# Controlling a Program's Execution Flow: if

- “if” statement: `if (boolean expression) statement`
  - *statement* can be a **simple statement** - a single line of Java code terminating in a semicolon (;) - or a **compound statement** - i.e., a block of statements enclosed in curly braces ({ ... })

```
// Note: use == to test for equality
if (total == limit) j = 2;
```

```
// Braces are preferred, however.
if (total == limit) {
    j = 2;
}
```

# if, cont.

```
if (!done) {  
    x = 3;  
    y = 4;  
    System.out.println("foo!");  
}
```

# if, cont.

- An "if" statement may optionally be paired with an "else" statement (simple or compound)

```
if (!done) { ... }  
else { ... }           // compound statement
```

- These can be nested to any arbitrary depth

```
if (x < 0) {  
    if (z > 100) {  
        f = 0;  
    }  
    else {  
        if (r == 12) {  
            f = 2;  
        }  
    }  
}  
else {  
    y = y - 1;  
}
```

# if, cont.

- When we wish to make up a complex 'if' clause that involves multiple boolean expressions, we can use
  - **logical operators:** == (equal to), != (not equal to), <, >, <=, >=
  - **boolean operators:** && (and), || (or), or ! (not)to combine these

# if, cont.

```
// If x is a positive number AND y is a positive number ...  
if ((x > 0) && (y > 0)) ...
```

```
// If x is a negative number OR y is a negative number ...  
if ((x < 0) || (y < 0)) ...
```

```
// If both x AND y are negative, OR z is positive ...  
if (((x < 0) && (y < 0)) || (z > 0)) ...
```

```
// If it is NOT true that both x and y are negative ...  
if (!(x < 0) && (y < 0)) ...
```

# Execution Flow, cont.: for

- "for" statements:

```
for ( initialization statement;  
      boolean expression;  
      increment/decrement statement )  
      simple/complex statement to be performed
```

- Pseudocode:

Step 1: Execute the *initialization statement* once

Step 2: Perform the *boolean test*

If outcome of test is "true":

Perform "*statement to be performed*"

Perform *increment/decrement statement*

Repeat Step 2

Otherwise, if outcome of test is "false", we're done!

# for, cont.

```
// Simple:  
for (int j = 1; j <= 100; j++) System.out.println(j);
```

```
// or:  
for (int j = 1; j <= 100; j++)  
    System.out.println(j);
```

(can break along  
white space)

```
// Preferred:  
for (int j = 1; j <= 100; j++) {  
    System.out.println(j);  
}
```

Can have as many lines within a **block** as necessary to accomplish a particular goal

# for, cont.

```
// Print out the integer values from 1 to 4, and  
// compute a running total.
```

```
int sum = 0;
```

```
for (int i = 1; i <= 4; i++) {  
    System.out.println(i);  
    sum = sum + i;  
}
```



# for, cont.

```
// Nested:  
  
int max = 3;  
  
for (int i = 1; i <= max; i++) {  
    System.out.print(i + " - ");  
  
    for (int j = i; j <= max; j++) {  
        System.out.print(j);  
    }  
  
    System.out.println();  
}
```

# for, cont.

// Nested:

```
int max = 3;
```

```
for (int i = 1; i <= max; i++) {  
    System.out.print(i + " - ");  
    for (int j = i; j <= max; j++) {  
        System.out.print(j);  
    }  
    System.out.println();  
}
```

i = 1

---

**Output:**

1 -

# for, cont.

// Nested:

```
int max = 3;
```

```
for (int i = 1; i <= max; i++) {  
    System.out.print(i + " - ");  
    for (int j = i; j <= max; j++) {  
        System.out.print(j);  
    }  
    System.out.println();  
}
```

i = 1  
j = 1

---

**Output:**

1 - 1

# for, cont.

```
// Nested:
```

```
int max = 3;
```

```
for (int i = 1; i <= max; i++) {  
    System.out.print(i + " - ");  
    for (int j = i; j <= max; j++) {  
        System.out.print(j);  
    }  
    System.out.println();  
}
```

i = 1  
j = 2

---

**Output:**

```
1 - 12
```

# for, cont.

// Nested:

```
int max = 3;
```

```
for (int i = 1; i <= max; i++) {  
    System.out.print(i + " - ");  
    for (int j = i; j <= max; j++) {  
        System.out.print(j);  
    }  
    System.out.println();  
}
```

i = 1  
j = 3

---

**Output:**

1 - 123

# for, cont.

// Nested:

```
int max = 3;
```

```
for (int i = 1; i <= max; i++) {  
    System.out.print(i + " - ");  
  
    for (int j = i; j <= max; j++) {  
        System.out.print(j);  
    }  
}
```

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

```
}
```

i = 1  
j = ?

---

**Output:**

1 - 123 ↻

# for, cont.

// Nested:

```
int max = 3;
```

```
for (int i = 1; i <= max; i++) {  
    System.out.print(i + " - ");  
    for (int j = i; j <= max; j++) {  
        System.out.print(j);  
    }  
    System.out.println();  
}
```

i = 2

---

**Output:**

1 - 123

2 -

# for, cont.

```
// Nested:
```

```
int max = 3;
```

```
for (int i = 1; i <= max; i++) {  
    System.out.print(i + " - ");  
    for (int j = i; j <= max; j++) {  
        System.out.print(j);  
    }  
    System.out.println();  
}
```

i = 2  
j = 2

---

**Output:**

```
1 - 123  
2 - 2
```



# for, cont.

```
// Nested:
```

```
int max = 3;
```

```
for (int i = 1; i <= max; i++) {  
    System.out.print(i + " - ");  
    for (int j = i; j <= max; j++) {  
        System.out.print(j);  
    }  
    System.out.println();  
}
```

i = 2  
j = 3

---

**Output:**

```
1 - 123  
2 - 23
```

# for, cont.

// Nested:

```
int max = 3;
```

```
for (int i = 1; i <= max; i++) {  
    System.out.print(i + " - ");  
  
    for (int j = i; j <= max; j++) {  
        System.out.print(j);  
    }  
  
    System.out.println();  
}
```

i = 2  
j = ?

---

**Output:**

1 - 123

2 - 23 ↻

# for, cont.

// Nested:

```
int max = 3;
```

```
for (int i = 1; i <= max; i++) {  
    System.out.print(i + " - ");  
    for (int j = i; j <= max; j++) {  
        System.out.print(j);  
    }  
    System.out.println();  
}
```

i = 3

---

## Output:

1 - 123

2 - 23

3 -

# for, cont.

// Nested:

```
int max = 3;
```

```
for (int i = 1; i <= max; i++) {  
    System.out.print(i + " - ");  
    for (int j = i; j <= max; j++) {  
        System.out.print(j);  
    }  
    System.out.println();  
}
```

i = 3

j = 3

---

## Output:

```
1 - 123  
2 - 23  
3 - 3
```

# for, cont.

// Nested:

```
int max = 3;
```

```
for (int i = 1; i <= max; i++) {  
    System.out.print(i + " - ");  
  
    for (int j = i; j <= max; j++) {  
        System.out.print(j);  
    }  
}
```

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

```
}
```

i = 3

j = ?

---

**Output:**

1 - 123

2 - 23

3 - 3



# Execution Flow, cont.: while

- “while” statements

`while (boolean expression) statement to be executed`

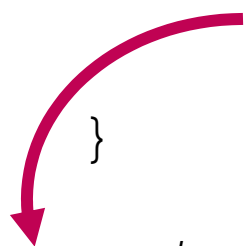
```
int i = 1;
int sum = 0;

// Print all integers between 1 and 4, and
// compute their total.
while (i < 5) {
    System.out.println(i);
    sum = sum + i;
    // Remember to increment i!
    i++;
}
```

# Execution Flow, cont.

- There are two ways to prematurely get out from within the middle of a 'for' or 'while' loop
  - First way: **break**

```
// Prints 0 1.  
for (int i = 0; i <= 4; i++) {  
    if (i == 2) break; // Terminates loop.  
    System.out.println(i);  
}
```




*next statement ...*

# Execution Flow, cont.

## - Second way: **continue**

```
// Prints 0 1 3 4.  
for (int i = 0; i <= 4; i++) {  
    if (i == 2) continue; // Terminates iteration.  
    System.out.println(i);  
}
```





# Case Sensitivity

- Reserved words must be spelled exactly as presented in this course, including upper/lower case usage
  - Baffling compilation errors may arise if you don't use proper case

```
For (int i = 0; i < 5; i++ { whatever }
```

^

**.class expected**

# Reserved Words

abstract	default	goto	null	synchronized
boolean	do	if	package	this
break	double	implements	private	throw
byte	else	import	protected	throws
case	extends	instanceof	public	transient
catch	false	int	return	true
char	final	interface	short	try
class	finally	long	static	void
const	float	native	super	volatile
continue	for	new	switch	while

(plus, built-in class names, which we'll learn about later ...)

# Block Structured Languages and Variable Scope

- Java (like C/C++) is a **block structured language**
- A 'block' of code is a series of zero or more lines of code enclosed within curly braces { ... }
  - A method, like the main method of our "Simple" program, defines a block
  - A class definition, like the "Simple" class as a whole, is also a block
  - Control flow statements also frequently involve defining blocks of code, aka compound statements

# Block Structured Languages and Variable Scope, cont.

- Blocks may be nested to any arbitrary depth
- The **scope** of a variable is defined as that portion of code for which a variable name remains defined to the compiler
  - Namely, from the point where it is first declared down to the closing (right) curly brace for the block of code that it was declared in

```

// Start of an outer block; no variables declared yet.
public class MyClass {
    static int x = 0;
    // x is now considered to be 'in scope'! Compiler will recognize x.
    // Start of a nested inner block (#1).
    public static void main(String[] args) {
        int y = 0;
        // Both x and y are now 'in scope'.
        // Start of a more deeply nested inner block (#2).
        if (some condition) {
            int z = 0;
            // x, y, and z are all in scope.
            x = y + z;
        } // end of nested block #2 - compiler forgets about z!
        // We can still manipulate x and y, because they are
        // both still in scope.
        y = x + 17;
    } // end of nested block #1 - compiler forgets about y!
    // We can still reference x, however, until we reach the
    // closing curly brace for MyClass.
    public void printX() {
        System.out.println(x);
    }
} // end of class

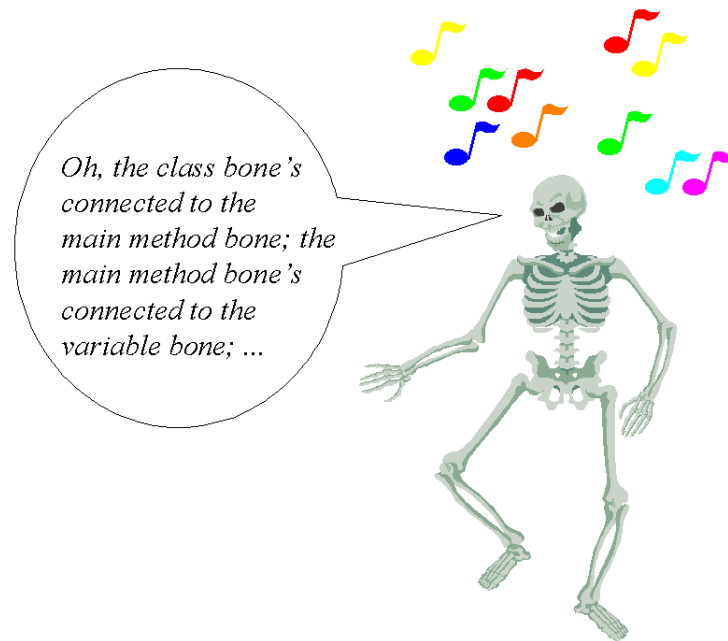
```

```
// Start of an outer block; no variables declared yet.
public class MyClass {
    static int x = 0;
    // x is now considered to be 'in scope'! Compiler will recognize x.
    // Start of a nested inner block (#1).
    public static void main(String[] args) {
        int y = 0;
        // Both x and y are now 'in scope'.
        // Start of a more deeply nested inner block (#2).
        if (some condition) {
            int z = 0;
            // x, y, and z are all in scope.
            x = y + z;
        } // end of nested block #2 - compiler forgets about z!
        // We can still manipulate x and y, because they are
        // both still in scope.
        y = x + 17;
    } // end of nested block #1 - compiler forgets about y!
    // We can still reference x, however, until we reach the
    // closing curly brace for MyClass.
    public void printX() {
        System.out.println(x);
    }
} // end of class
```

Proper indentation  
is important!

# Exercise #2

# The OO Building Blocks: Objects and Classes (Chapter 3)





# Software at its Simplest

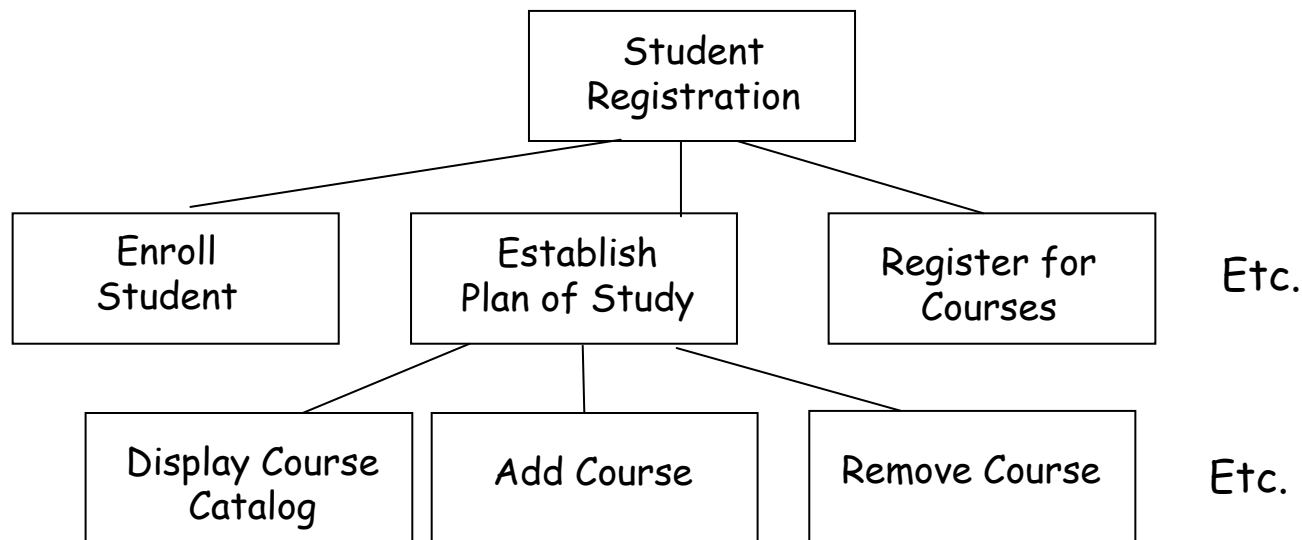
- Every software application consists of:
  - Data
  - Functions that operate on the data
- The pre-OO way of designing software was known as "(top-down) functional decomposition"

# Functional Decomposition

- With F.D., we started with a statement of the overall function that a system was expected to perform
  - E.g., "Student Registration"
- We then broke that function down into subfunctions
  - "Enroll Student"
  - "Establish Plan of Study"
  - "Register for Courses"
  - "Print Class Schedule"
  - "Print Student Roster"

# Functional Decomposition, cont.

- We kept subdividing functions into smaller and smaller pieces



- We then assigned **modules** to different programmers to implement, assembling them from the bottom up

# Functional Decomposition, cont.

- With this approach, data was an "afterthought"
  - Was passed around from one function to the next, like a car on an assembly line
  - Data structure had to be understood in MANY places throughout an application
  - If the data structure changed, there were major "ripple effects" (example: Y2K)
  - If data became corrupted after an application had been deployed, it was hard to pinpoint where (i.e., in what function) this had occurred

# Compared to the OO Approach

- As we'll see over the next several lectures, the object-oriented approach remedies the vast majority of these shortcomings
  - Data comes first, functions second
  - Data is encapsulated inside of objects
  - Data structure only has to be understood by the object it belongs to
  - If the data structure changes, there are virtually NO "ripple effects"
  - An object is responsible for ensuring the integrity of its own data
  - If data becomes corrupted, we can pretty much assume it was the object itself that did the corrupting!

# What is an Object?

- Before we talk about software objects, let's talk about real world objects in general
  - According to Merriam Webster's dictionary:  
*"(1) Something material that may be perceived by the senses; (2) something mental or physical toward which thought, feeling, or action is directed."*
  - The first part of this definition refers to objects as we typically think of them: as physical 'things' that we can see and touch, and which occupy space

# What is an Object?, cont.

- Let's think of some examples of **physical objects** that make sense in the general context of an academic setting:
  - The **students** who attend classes;
  - The **professors** who teach them;
  - The **classrooms** in which class meetings take place;
  - The **buildings** in which the classrooms are located;
  - The **textbooks** students useetc.

# What is an Object?, cont.

- There are also great many **conceptual objects** that play important roles in an academic setting; some of these are:
  - The **courses** that students attend;
  - The **departments** that faculty work for;
  - The **degrees** that students receive;etc.



# What is a Software Object?

- A **software object** is a software construct (module) that bundles together
  - **data** and
  - **functions**necessary to model the
  - **structure** and
  - **behavior**of a 'real-world' (physical or conceptual) object
- A building block of an object-oriented software system

# Data/Attributes

- If we wish to record information about a student, what data might we require?
  - Name
  - Student ID no.
  - Address
  - Major field of study, if they have declared one yet.
  - Grade point average (GPA)
  - Current course load
  - Transcript
- The data elements used to describe an object are referred to as the object's **attributes**
- A given attribute may be simple (GPA) or complex (transcript)

# State

- An object's attribute values, when taken collectively, are said to define the **state**, or condition, of the object
  - Is a student 'eligible to graduate' (a state)? Look at a combination of:
    - The student's transcript (an attribute), and
    - The list of courses they are currently enrolled in (a second attribute)

# Functions/Behavior

- A student's behaviors might include:
  - Enrolling in a course;
  - Dropping a course;
  - Choosing a major field of study;
  - Selecting a faculty advisor;
  - Telling you his or her *GPA* when asked;
  - Telling you whether or not he or she has taken a particular course

# Functions/Behavior, cont.

- It is a bit harder to think of an inanimate, conceptual object as having behaviors, but they do
- A course's behaviors might include:
  - Allowing a student to enroll;
  - Providing a list of all attending students;
  - Telling you which professor is teaching the course;
  - Telling you the classroom location in which it is being taughtand so forth

# Methods

- When we talk about software objects, we refer to an object's behaviors generically as **operations** and, when rendered in an OOPL, as its **methods**
  - A method is nothing more than a function -- one that is performed by a specific object, however
- In performing a method, an object typically accesses -- and optionally modifies -- its data (attribute values)
  - Telling you his or her GPA involves accessing the value of a student's 'GPA' attribute
  - Enrolling in a course updates the value of a student's 'course load' attribute

# What is a Class?

- A **class** is the programmatic way that we define the data structure and methods of all members in a group of similar objects
  - Attribute names and data types
  - The methods that each object belonging to the class is capable of performing
    - method names,
    - how to formally request them,
    - what 'behind the scenes' things an object has to do in accomplishing each one

# Example

- The Student class may declare:
  - 9 attributes

Attribute Name	(Java) Data Type
name	String
studentId	String
birthdate	Date *
address	String
major	String
gpa	float
advisor	???
courseLoad	???
transcript	???

\* *Date is another built-in Java data type*



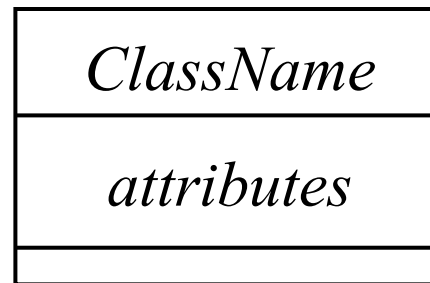
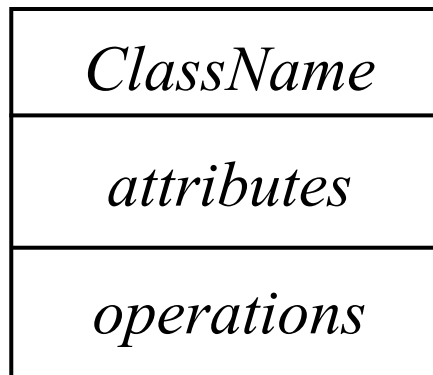
# Example, cont.

- 5 methods
  - registerForCourse
  - dropCourse
  - chooseMajor
  - changeAdvisor
  - printTranscript
- We use the term **feature** to refer collectively to attributes and methods
  - 9 attributes + 5 methods = 14 features
- Microsoft coined the term **property** to refer to the same notion
  - Purposely blurs the distinction between attributes and methods (we'll revisit this later)

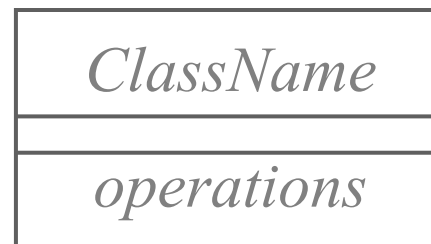
# Naming Conventions

- Name classes starting with an upper case letter
- Use mixed case to construct "readable" names from multiple word phrases
- Examples:
  - Student
  - Course
  - ScheduleOfClasses
- Name attributes and methods starting with a lower case letter: `studentIdNo`, `chooseMajor()`

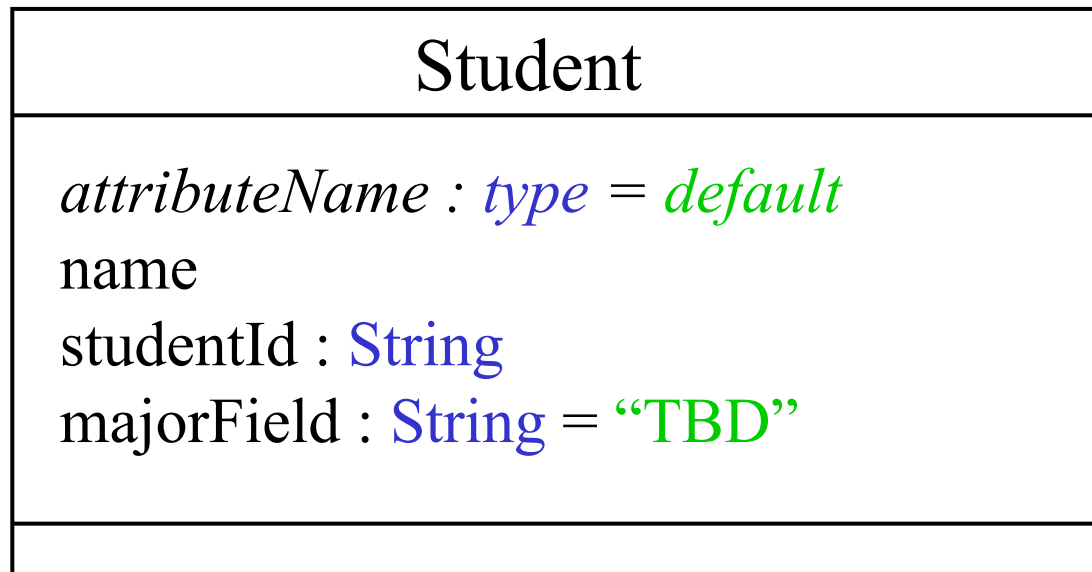
# Classes as UML Constructs



*uncommon:*



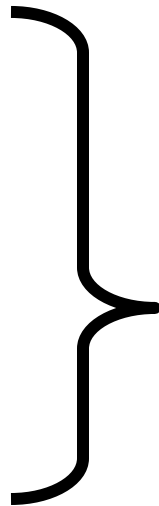
# Attributes in the UML



# Classes as Java Code

```
// File: Student.java
```

```
public class Student {  
    // Attributes.  
    String name;  
    String studentID;  
    String major;  
    float gpa;  
    String birthdate;  
    String address;  
    // ??? advisor;  
    // ??? courseLoad;  
    // ??? transcript;
```



Attributes are variable  
declarations at the  
class level of scope

Student
name : String studentID : String major : String gpa : float <i>etc.</i>

```
    // Methods will go here ... details to follow!  
    public boolean registerForCourse(...) { ... }  
}
```

# Instantiation

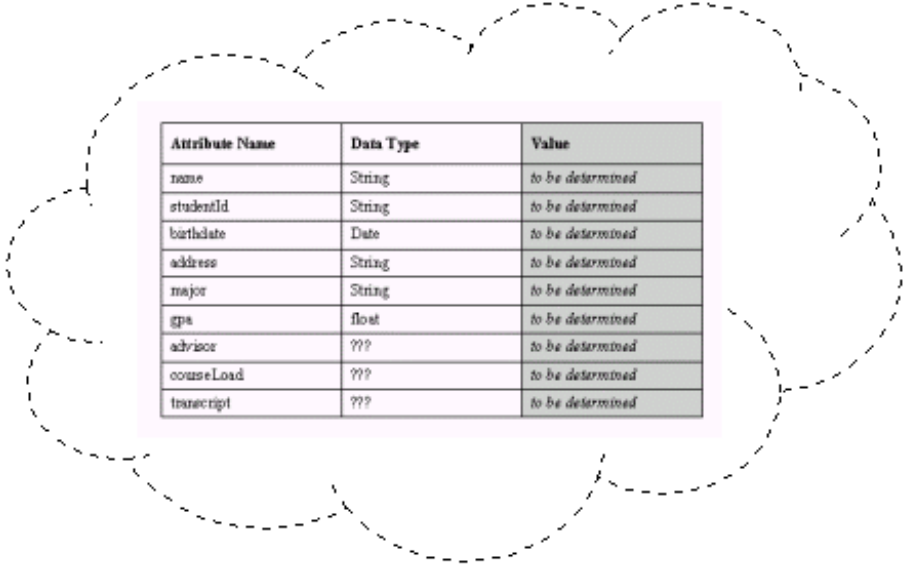
- A class definition may be thought of as a template for creating software objects
  - Like a cookie cutter or a rubber stamp
- The term **instantiation** is used to refer to the process by which an object is constructed based upon a class definition
  - Allocate a prescribed data structure in memory to house the attributes of a new object, and
  - Associate a certain set of behaviors with that object
- Another way to refer to an object, then, is as an **instance** of a particular class

# Classes vs. Objects

- An object's data structure is fixed based on the class it belongs to
  - The object fills in the data structure by providing values for its attributes
- An object can only do those things for which methods have been defined by the object's class
  - In this respect, an object is like an appliance

# Classes vs. Objects

*The Student class defines a template ...*

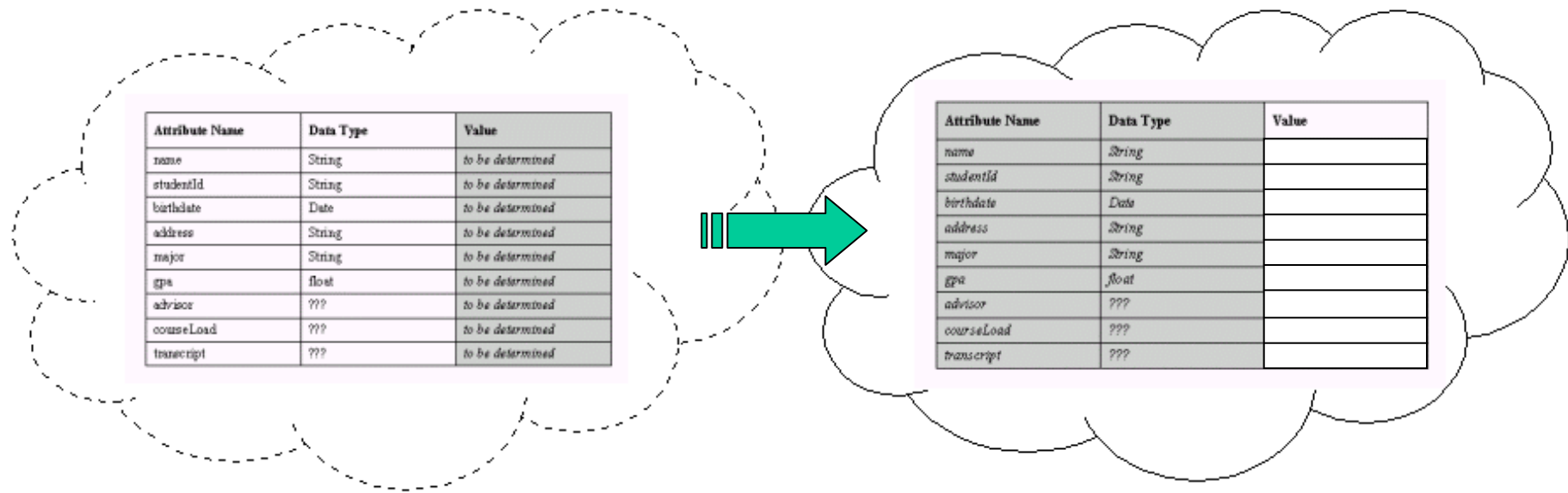


Attribute Name	Data Type	Value
name	String	to be determined
studentId	String	to be determined
birthdate	Date	to be determined
address	String	to be determined
major	String	to be determined
gpa	float	to be determined
advisor	???	to be determined
courseLoad	???	to be determined
transcript	???	to be determined



# Classes vs. Objects

*The Student class defines a template ...*



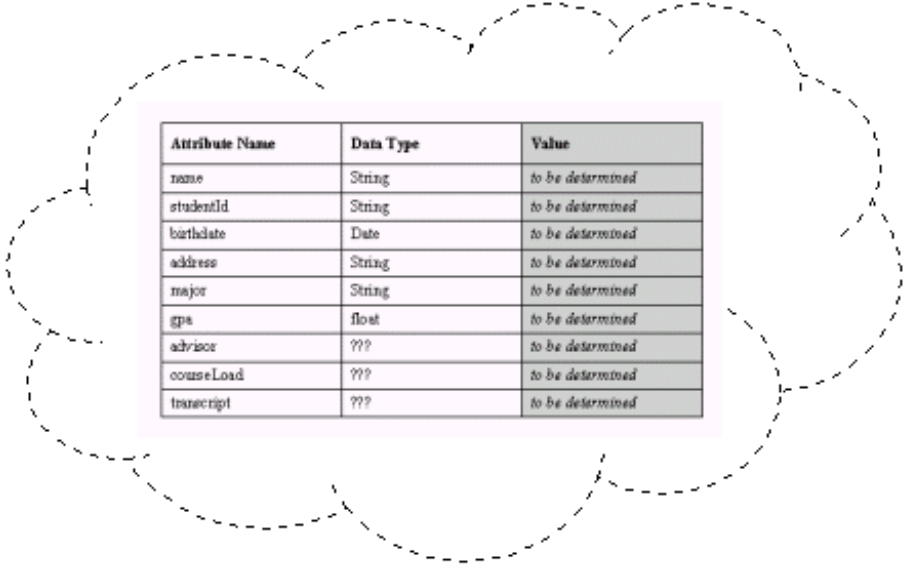
Attribute Name	Data Type	Value
name	String	to be determined
studentId	String	to be determined
birthdate	Date	to be determined
address	String	to be determined
major	String	to be determined
gpa	float	to be determined
advisor	???	to be determined
courseLoad	???	to be determined
transcript	???	to be determined

*... which gets stamped out in memory when a new object is instantiated.*

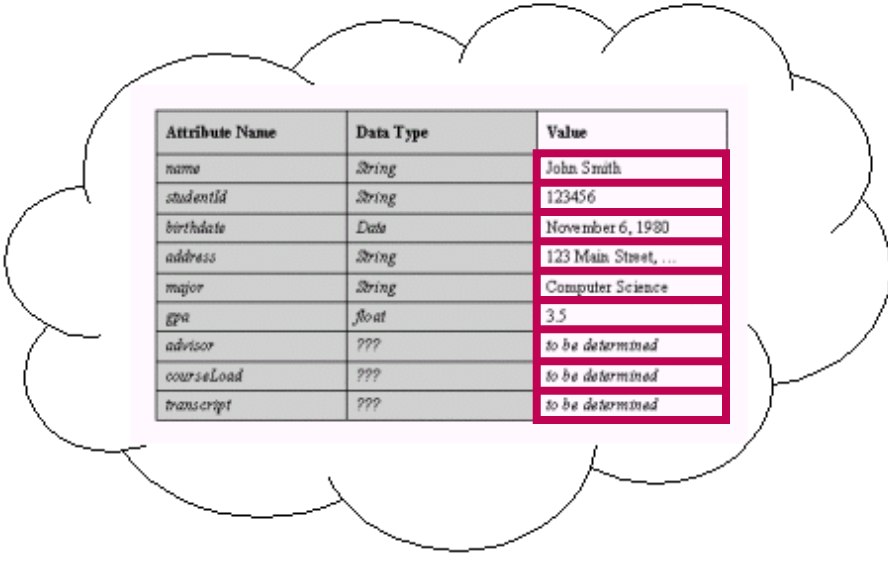
Attribute Name	Data Type	Value
name	String	
studentId	String	
birthdate	Date	
address	String	
major	String	
gpa	float	
advisor	???	
courseLoad	???	
transcript	???	

# Classes vs. Objects

*Each student object then fills in its own unique attribute values.*



Attribute Name	Data Type	Value
name	String	to be determined
studentId	String	to be determined
birthdate	Date	to be determined
address	String	to be determined
major	String	to be determined
gpa	float	to be determined
advisor	???	to be determined
courseLoad	???	to be determined
transcript	???	to be determined



Attribute Name	Data Type	Value
name	String	John Smith
studentId	String	123456
birthdate	Date	November 6, 1980
address	String	123 Main Street, ...
major	String	Computer Science
gpa	float	3.5
advisor	???	to be determined
courseLoad	???	to be determined
transcript	???	to be determined

# Variable Declarations and "Built-In" Data Types

- In a non-OOPL, the statement `int x;` is a **declaration** that variable "x" is an int(eger), one of Java's simple, "built-in" data types
- What does this really mean?
  - "x" is a symbolic name that represents an integer value which is stored as a 32 bit integer somewhere in memory -- we don't care where ...
  - The "thing" named "x" understands how to respond to a number of different arithmetic and logical operations as defined for the int data type
  - When we want to operate on this particular int. value, we refer to "x" instead: `if (x < 3) x = x + 1;`

# Behind the Scenes

- With any compiled program -- object oriented or otherwise -- we declare variables to represent data values

```
int x;
```

- The compiler automatically maps our variable to a memory address/location via a **symbol table**

# Behind the Scenes, cont.

Symbol Table:

<b>x</b>	<b>int</b>	<b>18024</b>
----------	------------	--------------

**int x;**

Program Memory:

...	
18023	
18024	
18025	
18026	
18027	
...	



# Behind the Scenes, cont.

Symbol Table:

x	int	18024
---	-----	-------

**x = 3;**

Program Memory:

...	
18023	
18024	<b>3</b>
18025	
18026	
18027	
...	

# Behind the Scenes, cont.

Symbol Table:

x	int	18024
---	-----	-------

**x = x + 1;**

Program Memory:

...	
18023	
18024	<b>4</b>
18025	
18026	
18027	
...	

# Classes as Abstract Data Types

- In an OOPL, we can define a class such as Student and then declare a variable such as:  
`Student y;`
- What does this really mean?
  - "y" is a symbolic name that refers to a Student object that is stored as a "chunk" of data somewhere in memory -- we don't care where ...
  - The "thing" that we have named "y" understands how to respond to a number of different service requests that have been defined by the Student class
  - Whenever we want to operate on this particular object, we refer to "y" instead:

```
if (y.getAge() >= 21) y.drinkBeer(); // ;o)
```



# Abstract Data Types, cont.

- Just as int is referred to as a simple, or built-in, data type, we can refer to a class such as Student as an **abstract data type (ADT)**: that is, a **user-defined data type** that specifies structure as well as behavior
  - It is called "abstract" because a class is an abstraction of real-world objects that reflects only relevant details
  - And, because "y" is a variable that *refers to* an instance (object) of the class Student, "y" may be alternatively referred to as either a **reference variable** or sometimes just as a **reference**

# Instantiating Objects: A Closer Look

- Different OO languages differ in terms of when an object is actually conceived
- In Java, when we declare a variable to be of a user-defined type - e.g., `Person p;` -- we haven't actually created an object in memory yet
  - Reference variable "p" merely has the *potential* to be a reference to a Person object
  - Until we initialize y by assigning it a specific object to refer to, p is said to have the value `null`

# Instantiating Objects, cont.

- We must take the distinct step of using the `new` operator to actually carve out a brand new object in memory:

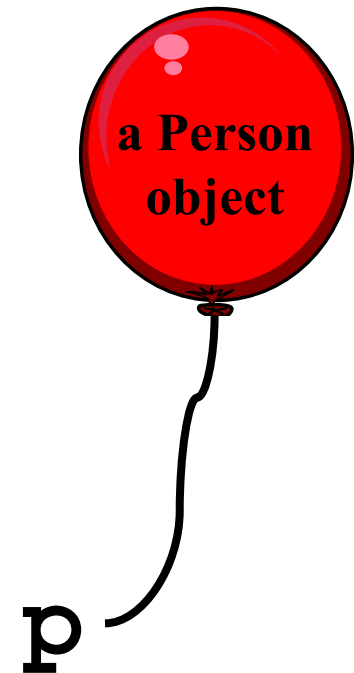
```
p = new Person();
```

- Its data structure in memory is mandated by the Student class definition
- We can combine the two steps - declaring a reference variable and actually instantiating an object for that variable to refer to - into a single line of code:

```
Person p = new Person();
```

# Object References

- Think of a newly created object as a helium balloon, and a reference variable as holding onto a string that is tied to the balloon
  - We refer to the object in our application code with the variable name "p"
  - We often use the slang term "handle", as in "p maintains a 'handle' on a Person object"
  - For those of you who know C/C++, a reference is a thinly veiled pointer ...



# Behind the Scenes, cont.

Symbol Table:

x	int	18024
<b>p</b>	<b>Person</b>	<b>18025</b>

**Person p;**

Program Memory:

...	
18023	
18024	
18025	
18026	
18027	
...	

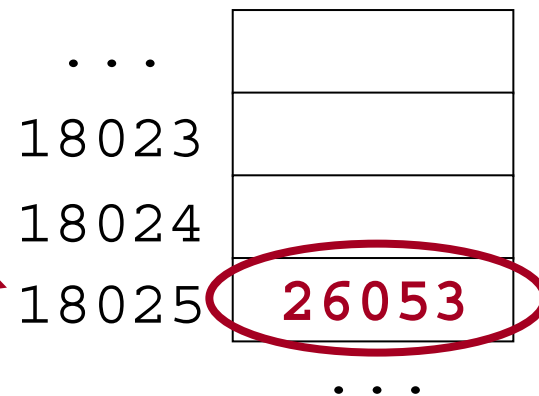


# Behind the Scenes, cont.

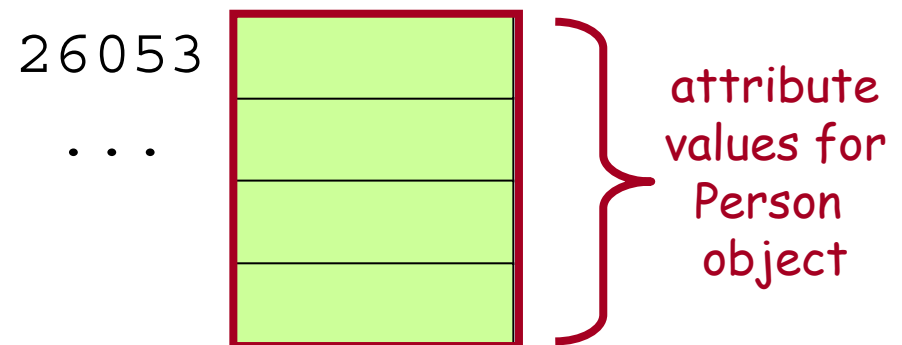
Symbol Table:

x	int	18024
p	Person	18025

Program Memory:



**p = new Person();**



# Behind the Scenes, cont.

Symbol Table:

x	int	18024
p	Person	18025

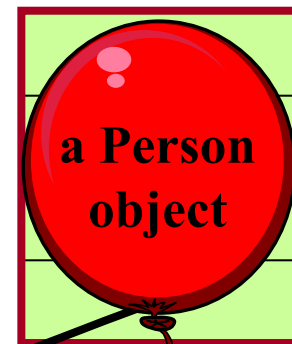
Program Memory:

...	
18023	
18024	
18025	26053

...

26053

...



# Instantiating Objects, cont.

- Another way to initialize a reference variable is to hand it a **preexisting** object: that is, an object ("helium balloon") whose handle ("string") is already being held by some **other** reference variable
- Let's look at an example involving Student objects ...



# Instantiating Objects, cont.

```
public static void main(String[] args) {  
    // We instantiate our first Student object.  
    Student x = new Student();  
  
    // We declare a second reference, but do not  
    // instantiate a second object.  
    Student y;
```

# Instantiating Objects, cont.

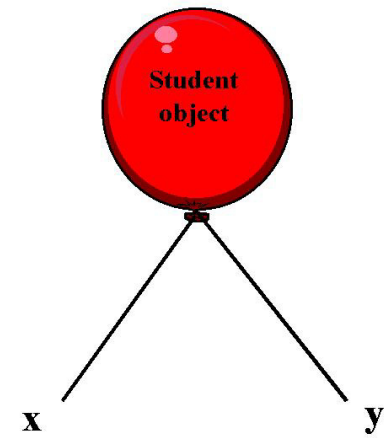
```
public static void main(String[] args) {  
    // We instantiate our first Student object.  
    Student x = new Student();  
  
    // We declare a second reference, but do not  
    // instantiate a second object.  
    Student y;  
  
    // We pass y a 'handle' on the same object  
    // that x is holding (x continues to hold  
    // onto it, too).  
    y = x;
```

# Instantiating Objects, cont.

```
// We instantiate our first Student object.  
Student x = new Student();
```

```
// We declare a second reference, but do not  
// instantiate a second object.  
Student y;
```

```
// We pass y a 'handle' on the same object  
// that x is holding (x continues to hold  
// onto it, too).  
y = x;
```



- The conceptual outcome: two 'strings' tied to the same 'balloon' - that is, two reference variables referencing the same object in memory

## Behind the Scenes

## Symbol Table:

x	Student	18025
y	Student	18047

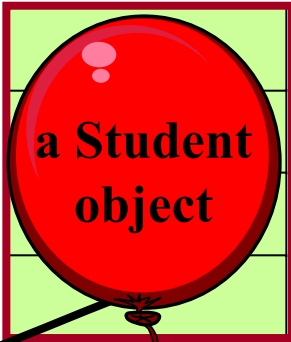
## Program Memory:

...	
18047	26053
<del>18024</del>	
18025	26053

• • •

26053

• • •

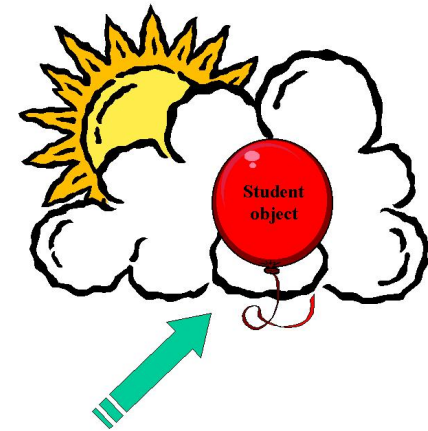


# 'Handles' on Objects

- We therefore see that the same object can have many reference variables holding onto it
- Conversely, any one reference variable can only hold onto *one* object at a time
  - To grab onto a new object 'handle' means that a reference variable must let go of any previous object 'handle' it was holding onto

# 'Handles' on Objects, cont.

- We therefore see that the same object can have many reference variables holding onto it
- Conversely, any one reference variable can only hold onto *one* object at a time
  - To grab onto a new object 'handle' means that a reference variable must let go of any previous object 'handle' it was holding onto
  - If there comes a time when *all* references holding onto a particular object have released the object's handle, then the object is no longer accessible to the program



# 'Handles' on Objects, cont.

```
// We instantiate our first Student object.  
Student x = new Student();  
  
// We declare a second reference, but do not  
// instantiate a second object.  
Student y;  
  
// We pass y a 'handle' on the same object  
// that x is holding.  
y = x;  
  
// We now declare a third reference and  
// instantiate a second Student object.  
Student z = new Student();
```

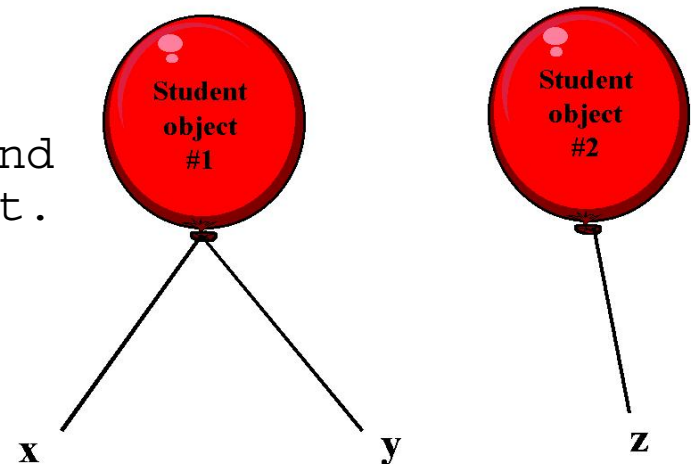
# 'Handles' on Objects, cont.

```
// We instantiate our first Student object.  
Student x = new Student();
```

```
// We declare a second reference, but do not  
// instantiate a second object.  
Student y;
```

```
// We pass y a 'handle' on the same object  
// that x is holding.  
y = x;
```

```
// We now declare a third reference and  
// instantiate a second Student object.  
Student z = new Student();
```



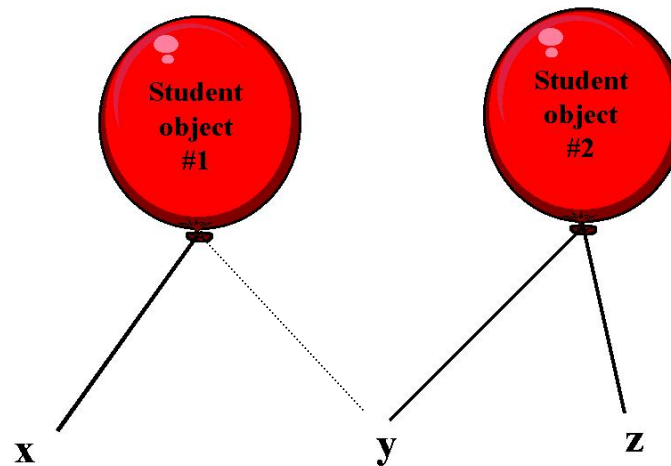


# 'Handles' on Objects, cont.

```
// y now lets go of the first Student object  
// and grabs onto the second.  
y = z;
```

# 'Handles' on Objects, cont.

```
// y now lets go of the first Student object  
// and grabs onto the second.  
y = z;
```



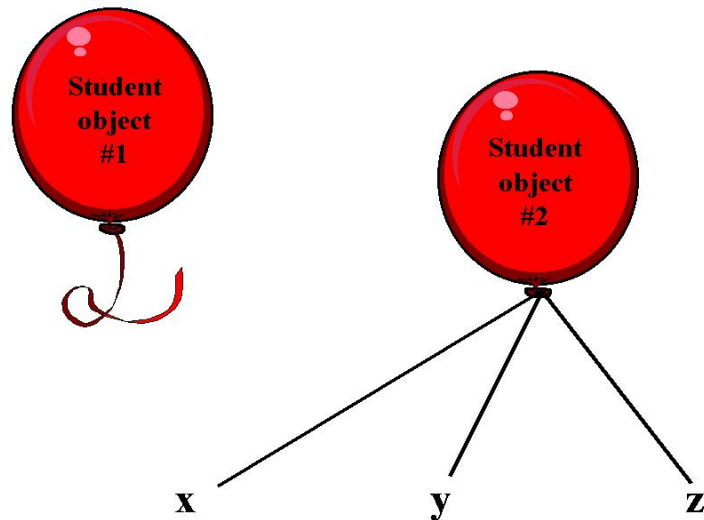
# 'Handles' on Objects, cont.

```
// y now lets go of the first Student object  
// and grabs onto the second.  
y = z;  
  
// Finally, x lets go of the first Student  
// object, and grabs onto the second, as well.  
x = z;
```

# 'Handles' on Objects, cont.

```
// y now lets go of the first Student object  
// and grabs onto the second.  
y = z;
```

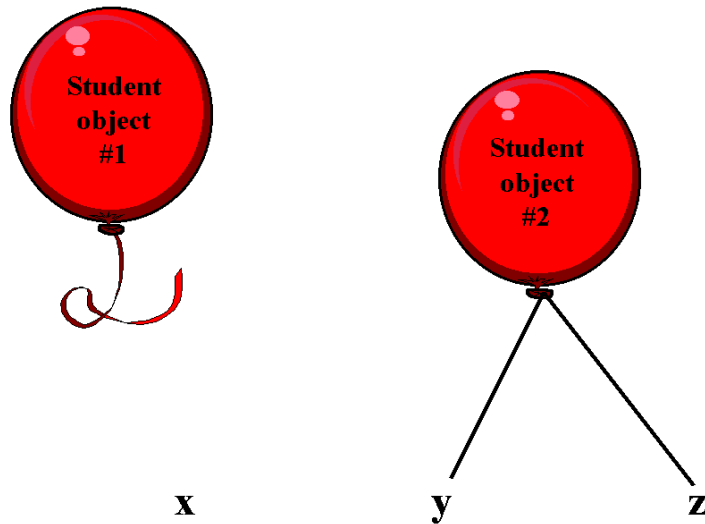
```
// Finally, x lets go of the first Student  
// object, and grabs onto the second, as well.  
x = z;  
// The first Student object is now lost to  
// the program.
```



# 'Handles' on Objects, cont.

// Another way to get a reference to let go of an object:

```
x = null;
```

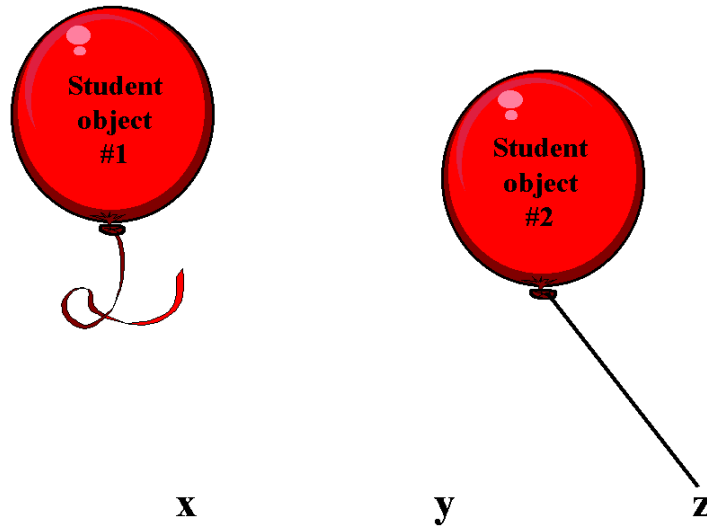


# 'Handles' on Objects, cont.

// Another way to get a reference to let go of an object:

```
x = null;
```

```
y = null;
```



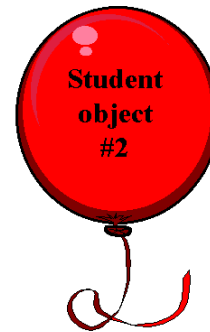
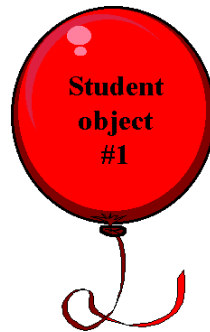
# 'Handles' on Objects, cont.

// Another way to get a reference to let go of an object:

```
x = null;
```

```
y = null;
```

```
z = null;
```

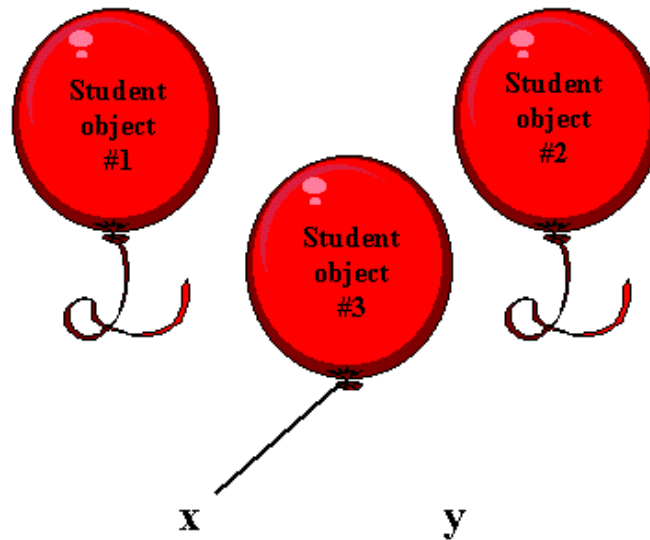


# 'Handles' on Objects, cont.

```
x = null;  
y = null;  
z = null;
```

```
// Whenever we see the "new" operator, we know that a  
// brand new object has been created.
```

```
x = new Student();
```





# References vs. Pointers

- For those of you familiar with the concept of pointers, from languages such as C and C++:
  - References are similar to pointers, in that they refer to memory locations/addresses where data (objects) are stored
  - References differ from pointers, in that they cannot be manipulated arithmetically the way that pointers can

# Garbage Collection

- If all of an object's handles are dropped, it might seem as though the memory that the object occupies is permanently 'wasted'
  - In a language like C++, this is indeed the case, unless programmers explicitly 'recycle' the memory of an object before all of its handles are dropped
  - In Java, on the other hand, there is a utility called the **garbage collector** built into the JVM which automatically recycles 'lost' objects' memory for us

# Garbage Collection, cont.

- If there are no remaining references to an object, it becomes a **candidate** for garbage collection
  - The garbage collector doesn't immediately recycle the object
  - Runs whenever the JVM determines that the application is getting low on free memory
  - So, for some period of time, the 'orphaned' Student object will still exist in memory - we merely won't have any handles with which to reach it

# Garbage Collection, cont.

- There is a way to explicitly request garbage collection:

```
Runtime.getRuntime().gc();    // nothing to import -  
                                // the Runtime class is  
                                // defined in java.lang
```

but, even then, the precise moment of *when* garbage collection occurs is out of the programmer's control, as is which and how many of the eligible objects will be collected

# Garbage Collection, cont.

- The inclusion of garbage collection in Java has virtually eliminated memory leaks
- Note that it is still possible for the JVM itself to run out of memory, however, if you maintain too many handles on too many objects
  - The amount of memory allocated to the JVM when it first starts up is command-line driven
- A Java programmer cannot be totally oblivious to memory management - it is just less error prone than with C/C++

# Naming Conventions, Revisited

- Name object references starting with a lower case letter
- Use mixed case to construct "readable" names from multiple word phrases
- Examples:
  - Student aStudent;
  - Professor departmentChair;

# Objects as Attributes

- When we first discussed the Student class, we stated that some of the attributes could be represented by built-in data types, whereas the types of a few others (advisor, courseLoad, and transcript) were left undefined
- Let's now put what we've learned about abstract data types to good use:
  - Rather than declaring the advisor attribute as simply a `String` representing the advisor's name ...
  - We'll declare it to be of an abstract data type - namely, type `Professor`, another user-defined type

# Objects as Attributes, cont.

Student attributes:

Attribute Name	Data Type
name	String
studentID	String
birthdate	Date
address	String
major	String
gpa	float
<b>advisor</b>	<b>Professor</b> ←
courseLoad	???
transcript	???

String

(we'll leave the others undefined for the time being)



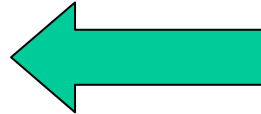
# Objects as Attributes, cont.

```
// File: Student.java

public class Student {
    // Attributes.

    String name;
    String studentID;
    String birthdate;
    String address;
    String major;
    float gpa;
    Professor advisor;
    // ??? courseLoad;
    // ??? transcript;

    // Methods go here ... stay tuned! :o)
}
```



# Objects as Attributes, cont.

Professor attributes:

Attribute Name	Data Type
name	String
employeeID	String
birthdate	Date
address	String
worksFor	String (or Department)
<b>studentAdvisee</b>	<b>Student</b> ←
teachingAssignments	???

String

# Abstraction, Revisited

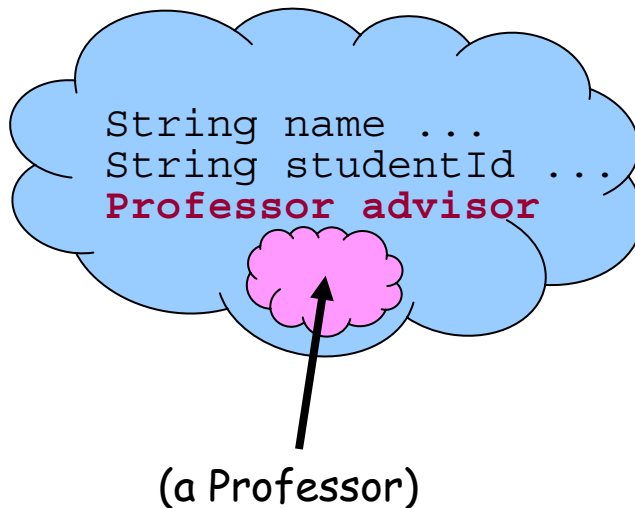
- Remember: decisions of inclusion vs. elimination must be made within the context of the overall **domain** of the future system
  - When representing a person, is his/her eye color important? How about his/her genetic profile? Salary? Hobbies?
  - Any of these features may be relevant or irrelevant, depending on the system to be developed

# Exercise #3

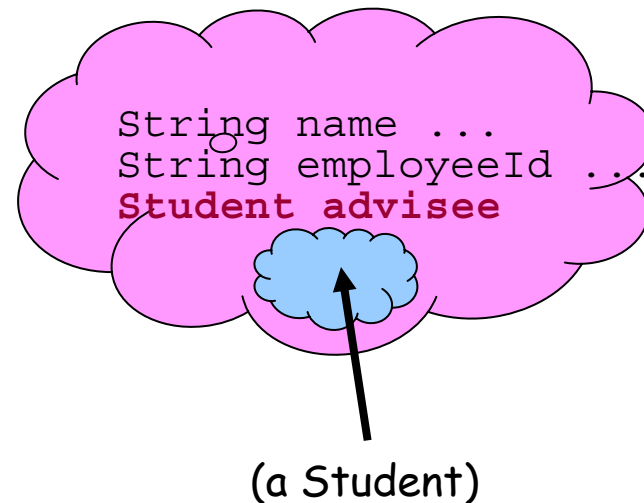
# Nested Objects?

- It might seem as though implementing objects as attributes would mean that a Student object has a Professor object "nested" inside, and vice versa ...

(a Student)

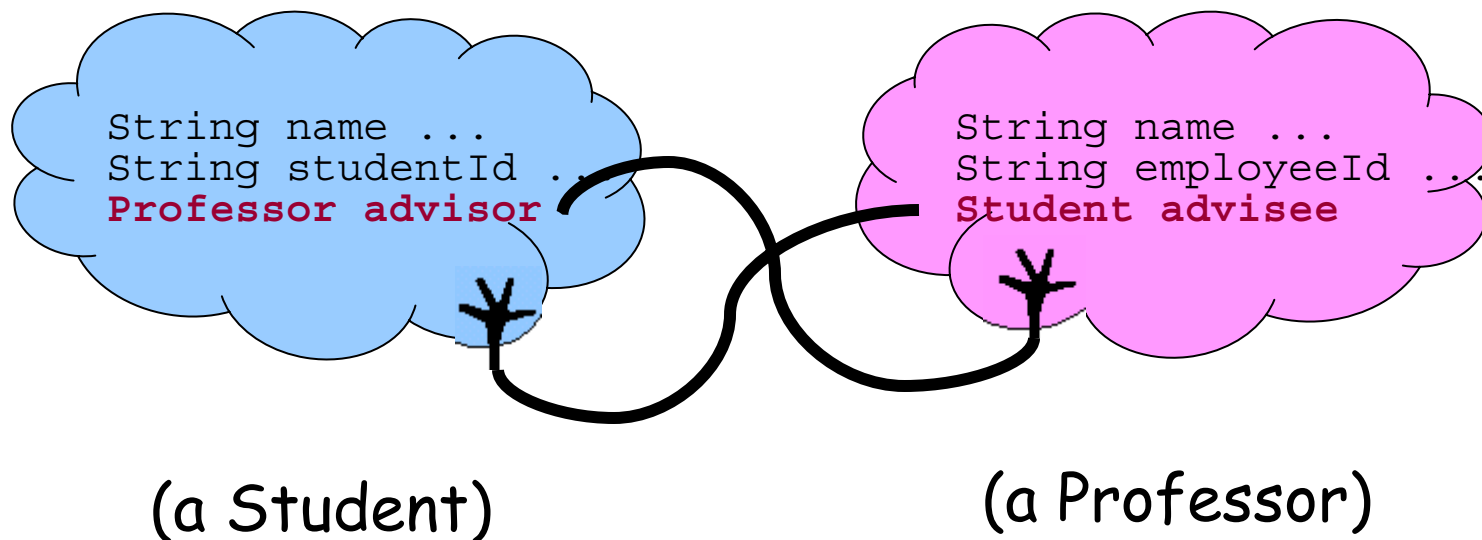


(a Professor)



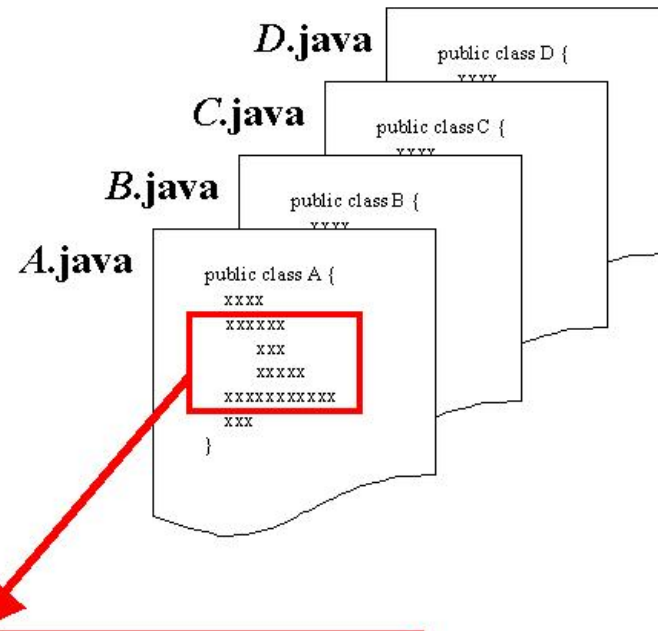
# Object References

- But in reality, each object only exists once in memory; objects store references to one another, so that they can find (refer to) one another as needed *(memory sketch)*



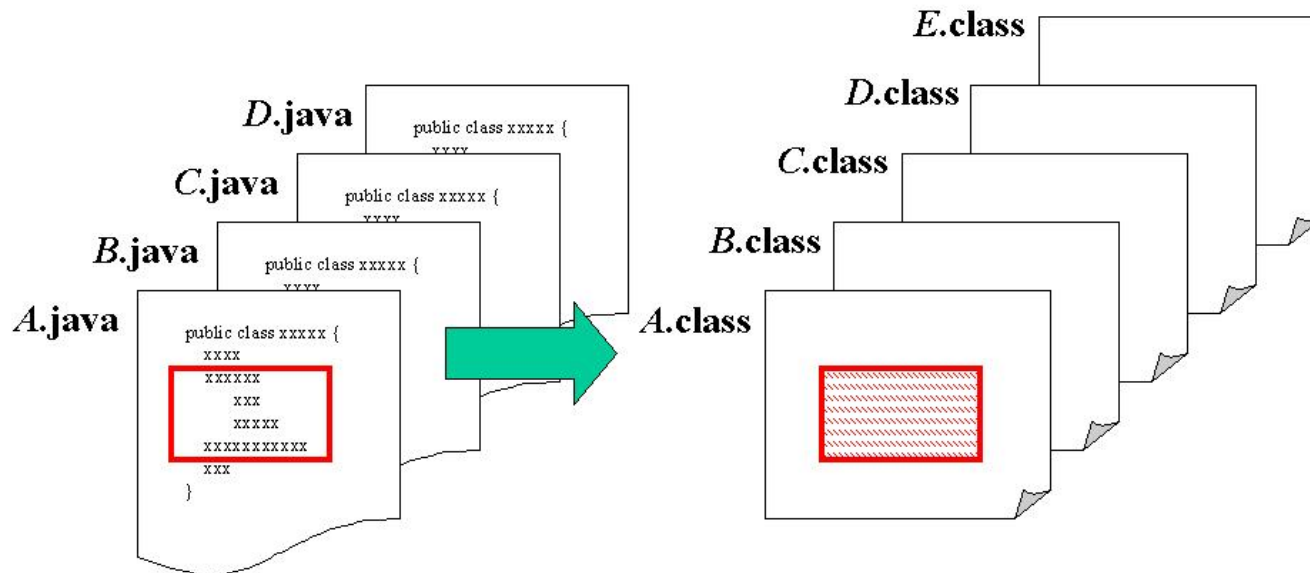
# Anatomy of an Application

*A Java program consists of many .java source files, each containing one (or more) class definitions; only one such file contains the “official” main method used to drive the program.*



```
public static void main(String[ ] args) {  
    xxxx  
    xxxxxxxxx  
    xxxxxxxx  
    ...  
}
```

# Anatomy, cont.



*Each properly written .java file, when compiled, generates one or more .class byte code files; again, only one such file contains the “official” main method used to drive the program.*



# Assigning Attribute Types

- Noteworthy points about the Professor class:
  - Having an attribute like `studentAdvisee` that can only track a single Student object may be limiting
  - We'll discuss techniques for handling this when we talk about **collections**, which will also be relevant to:
    - the `teachingAssignments` attribute of Professor
    - the `courseLoad` and `transcript` attributes of Student
  - The `worksFor` attribute can be declared as either:
    - A simple String representing the department name ("MATH")
    - A reference to a Department object

# Attribute Types, cont.

- The decision of whether or not we need to invent an abstract data type to represent a particular real-world concept is not always clear-cut
- Pose these questions:
  - Is the concept central to our model overall?
  - Does the concept warrant attributes and behaviors of its own?

If the answer to either or both is "yes", then it most likely warrants its own class

# Composite Classes

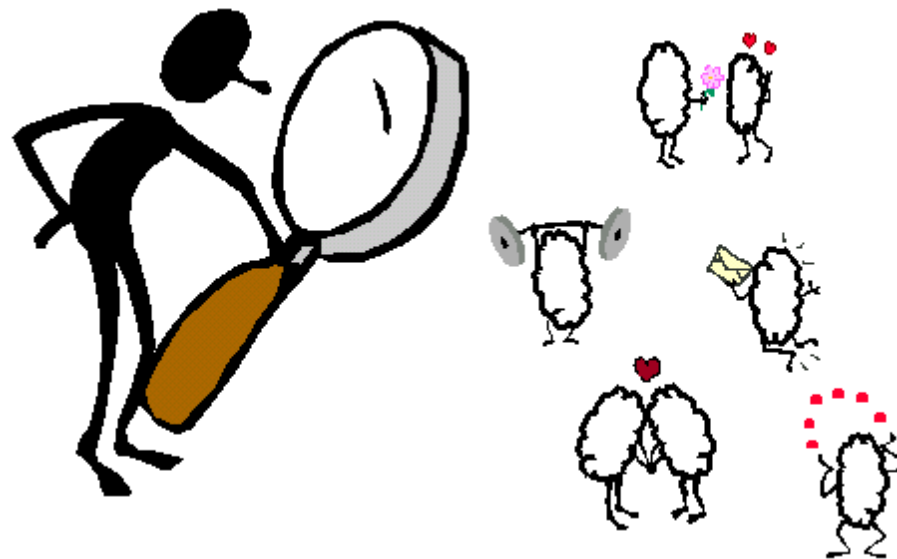
- Whenever we have a class, such as Student, in which one or more of the attributes are references to other objects, we refer to the class as a **composite class**
- The number of levels to which objects can be conceptually 'bundled' inside one another is endless, and so composite classes enable us to model very sophisticated real world concepts
- As it turns out, most 'interesting' classes are composite classes

# Three Features of an OOPL

- ✓ (Programmer creation of) Abstract Data Types (ADTs)
  - The ability to define classes which encapsulate data and behavior, and ...
  - ... to instantiate objects based on these classes
- Inheritance
- Polymorphism

# Object Behaviors

## (Chapter 4)



# Requesting Object Services

- One way to think of an object's methods are as **services** that can be requested of the object by the application as a whole
- In order to request some service of an object 'x', we need to know the specific language with which to communicate with x
  - We need to be clear as to exactly which of x's methods/services we want it to perform
  - Depending on the service request, we may need to give x some additional information so that x knows exactly how to proceed
  - x in turn needs to know whether we expect it to report back the outcome of what it has been asked to do

# An Example

- With respect to household chores, a Person is capable of:
  - mowing the lawn
  - taking out the trash
  - washing the car
- We decide that we want our teenaged sons Larry, Moe, and Curley, to do each of these three chores



## An Example, cont.

- How would we ask them to do this?
  - "Please\* wash the car."
  - "Please take out the trash."
  - "Please mow the lawn."

\* optional! ;o)



## An Example, cont.

- How would we ask them to do this?
  - "Please wash the car (the Camry)."  clarification
  - "Please take out the trash."
  - "Please mow the lawn, and let me know if you see any crabgrass."  reporting back

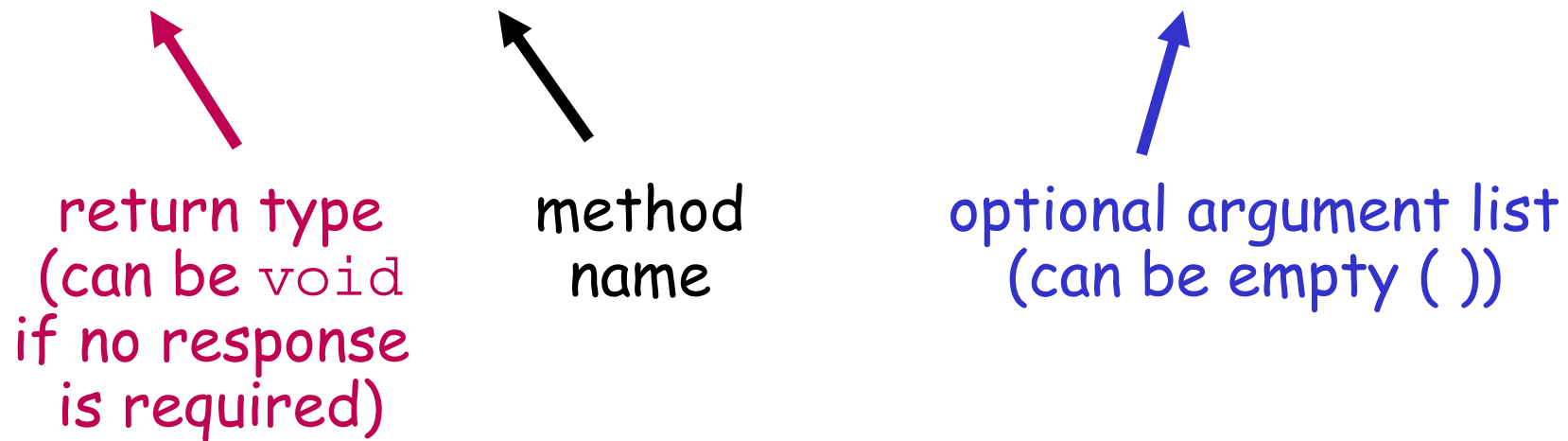
# Method Headers

- We take care of communicating these 3 aspects of each method by defining a **method header**
- A formal specification of how that method is to be invoked -- it consists of:
  - A method's **name**;
  - An optional list of **arguments** (names and types of variables) to be passed to the method, enclosed in parentheses; and
  - A method's **return type** - that is, the data type of the information that is going to be passed back, if any, when the method is finished executing

# Method Headers, cont.

- Example of a Student method:

boolean registerForCourse (String courseId, int sectionNumber)



(for folks familiar with C/C++, this is nothing more than a function specification)

# Method Headers, cont.

- Other examples:

```
boolean isHonorsStudent()  
           int getAge()  
void changeName(String newName)
```

return type    method name    (optional) argument list

- We declare a return type of `void` when we don't need any data to be passed back to us

# Argument Signatures

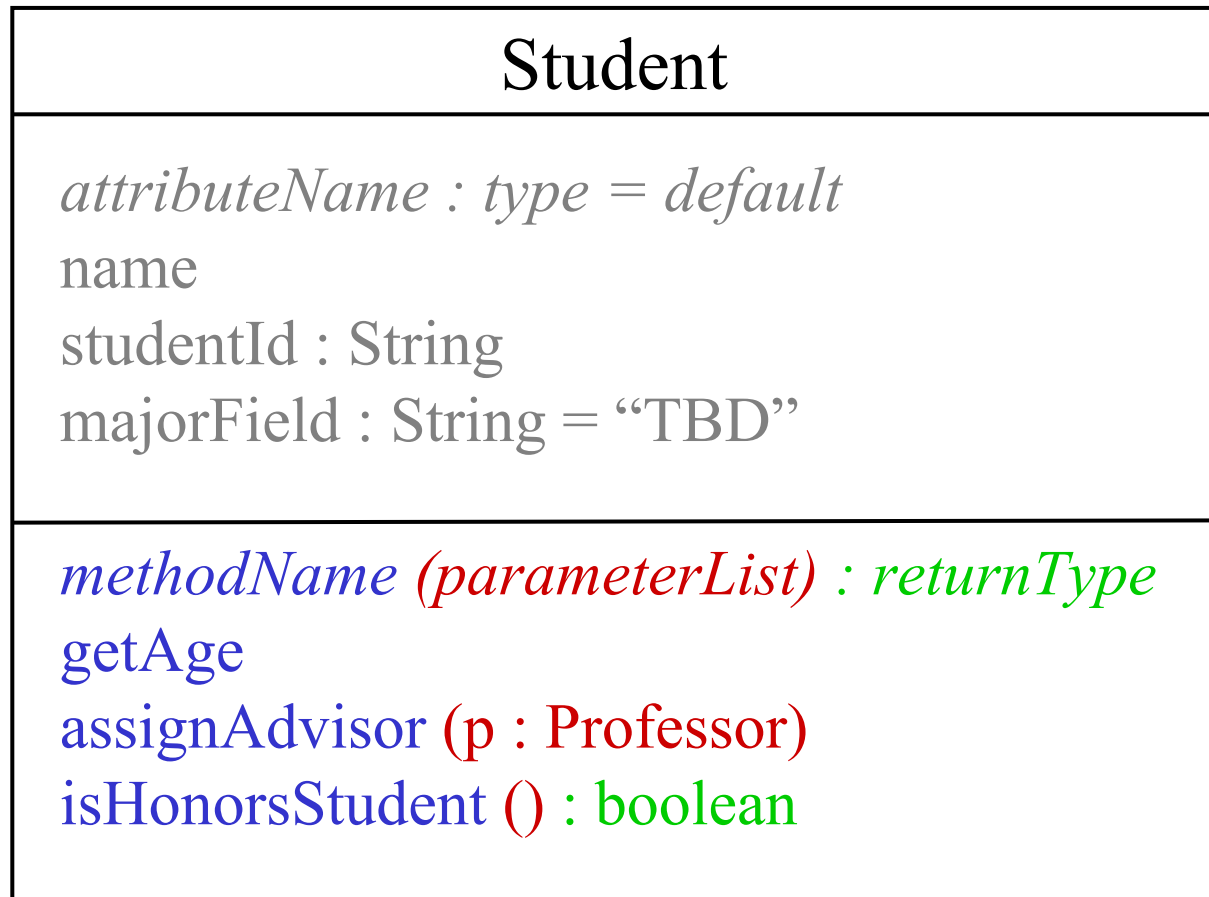
- The term **argument signature** refers to the order, types, and number of arguments comprising a method signature, but not their specific names
- Method signature:

```
boolean registerForCourse(String courseId, int sectionNumber)
```

has argument signature: (String, int)

The diagram consists of two arrows. One arrow originates from the 'String' part of the argument signature '(String, int)' and points diagonally upwards and to the left towards the 'String' parameter in the method signature 'registerForCourse(String courseId, int sectionNumber)'. The second arrow originates from the 'int' part of the argument signature and points diagonally upwards and to the right towards the 'int' parameter in the method signature.

# Methods in the UML



# Method Bodies

- We must also program the internal details of how each method should behave when invoked - **this represents business logic**

```
public class Student {  
    // Attributes.  
    float gpa;  
    // other attributes would follow here...  
  
    // Method.  
    boolean isHonorsStudent() {  
        // All attributes are "in scope" for all methods.  
        if (gpa >= 3.5) return true;  
        else return false;  
        // "return" statement not needed for void methods  
    }  
}
```

# Method Bodies

- We must also program the internal details of how each method should behave when invoked - **this represents business logic**

```
public class Student {  
    // Attributes.  
    float gpa;  
    // other attributes would follow here...  
  
    // Method.  
    boolean isHonorsStudent() {  
        // Alternatively:  
        return (gpa >= 3.5);  
    }  
  
    // etc.
```



# Message Passing and Dot Notation

- To represent in code that a service is being requested of an object, we invoke a method on that object's reference variable using **dot notation**

```
Student x = new Student();
```

```
x.registerForCourse("MATH 101", 10);
```

← a "message"

- A period ("dot") separates the object reference variable from the method (function) call
- The object is the context for the method ... think of the notation `x.` as "talking to object x"

(for folks familiar with C/C++, this is nothing more than a function call ... **aimed at a particular object!**)

# Passing Messages

- Returning to our example of asking our teenaged sons to do various chores, we need to target each request for a particular son:
  - "Larry, please wash the car (the Camry)."
  - "Moe, please take out the trash."
  - "Curley, please mow the lawn, and let me know if you see any crabgrass."

# Passing Messages

- Now, let's turn this into Java code:
  - Person is a class
  - We have defined the following methods for Person:

```
void washTheCar(Car c)
void takeOutTheTrash()
boolean mowTheLawn()
```

# Passing Messages

- We have three Person objects/instances:

```
Person larry = new Person();
```

```
Person moe = new Person();
```

```
Person curley = new Person();
```

- While we're at it, let's define the Car class/abstraction:

```
Car camry = new Car();
```

- We send a message to each, indicating the service that we wish each of them to perform:

```
larry.washTheCar(camry);
```

```
moe.takeOutTheTrash();
```

```
curley.mowTheLawn();
```

# Capturing the Result of a Method Call

- In our earlier example, although we declared the `registerForCourse()` method to have a return type of `boolean`:

```
boolean registerForCourse(String courseId, int sectionNumber)
```

we didn't capture the returned value when we invoked the method:

```
x.registerForCourse("MATH 101", 10);
```

# Capturing the Result of a Method Call, cont.

- We can optionally catch the result of a non-void method when we invoke it:

```
boolean outcome;
```

```
outcome = x.registerForCourse("MATH 101", 10);
```

- We can also react to the result without bothering to capture it in a distinct variable:

```
// An "if" expression must evaluate to a boolean result.
```

```
// Note the use of ! (not).
```

```
if (!(x.registerForCourse("MATH 101", 10))) {  
    action to be taken if registration failed ...  
}
```

# Client Code

- We refer to any body of code that invokes a method on an object 'X' as **client code** relative to 'X'
  - E.g., the main method is considered to be client code relative to `Student s` in the example below:

```
public static void main(String[] args) {  
    Student s = new Student();  
    // details omitted ...  
  
    // Invoke a method on Student object s.  
    double x = s.getGpa();  
}
```

because `s` is performing a service that is of benefit to the main method

## (optional) A Word About Argument Names

- Even in a non-OOPL, the names we give to the arguments of a function have no relationship to the names of the variables that are passed into the function from elsewhere in our application

- A C function:

```
double average(double x, double y) {  
    return (x + y) / 2.0;  
} // as of this closing brace, x and y go out of scope
```

- Calling this function from a C program:

```
double a = 17.3;  
double b = 38.7;  
double c = average(a, b); // the values of a and b are  
                          // assigned to x and y
```



# (optional) Passing Arguments

- In the preceding example, the arguments to the average function are said to be **passed by value**
- When an argument is passed by value, changing its value in a function doesn't affect the value of the original variable in the main program

```
void double_it(int x) {  
    x = 2 * x;  
} // as of this closing brace, the compiler forgets about x
```

- Using this function from a C program:

```
int y = 3;  
double_it(y);  
// y in the main program is still equal to 3!
```

- To change y's value:

```
y = double_it(y);
```

## (optional) Passing Arguments, cont.

- In Java, simple data types are **passed by value**
- Abstract data types, on the other hand, are **passed by reference**

- We are, in effect, passing the address of where a given object is located in memory

```
boolean registerForCourse(Course c) { ... }
```

---

```
// In the main program:
```

```
Course x = new Course();
```

```
Student s = new Student();
```

```
s.registerForCourse(x);
```

- If we change the attribute values of Course object 'c' inside of the method, the same changes apply to 'x', because they are both referring to the same object

# Argument Names

- When inventing names for the arguments of your methods, there are several schools of thought:
  - Choose creative names that make the arguments self-documenting; examples:

```
boolean registerForCourse(String courseId, int sectionNumber)
```

```
boolean registerForCourse(String courseName, int secNo)
```

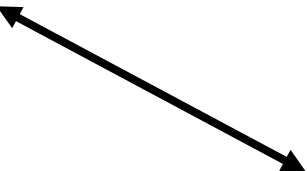
- Choose single letter names, but then make sure that the method is well documented; example:

```
// Arguments:  the name of the course and the section number.  
boolean registerForCourse(String c, int s)
```

# Argument Names, cont.

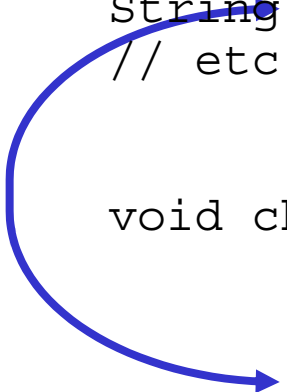
- AVOID argument names that duplicate attribute names of the class that the method belongs to:

```
public class Student {  
    String name;  
    // etc.  
  
    // BAD IDEA!  
    void changeName(String name) {  
        // Here, both references to name are to the  
        // locally scoped variable -- the attribute  
        // by this name doesn't get changed!  
        name = name;  
    }  
  
    // etc.  
}
```



# Argument Names, cont.

```
public class Student {  
    String name;  
    // etc.  
  
    void changeName(String name) {  
        // There is a workaround ... but, it is easy  
        // to forget to do this!  
        this.name = name;  
    }  
    // etc.  
}
```



# Accessing Attributes via Dot Notation

- In theory, we can also use dot notation to directly manipulate an object's internal attributes from client code:

```
Student x = new Student();  
x.name = "John Smith";
```

- However, classes often restrict access to some of their features (attributes in particular) -- such restriction is known as **information hiding**

# Information Hiding

- In a well designed OO application, a class publicizes
  - *what* it can do - i.e. the services it is capable of providing, or its method signaturesbut *hides* the internal details both of
  - *how* it performs these services (method bodies) and
  - the *data* (attributes) that it maintains in order to *support* these services
- Example: Yellow Pages ad for a dry cleaner

# Visibility

- We use the term **visibility** to refer to whether or not a particular feature (attribute/method) can be accessed from client code by applying dot notation to the object's reference variable
  - Public visibility:

```
public class Student {  
    public String name;  
    // etc.
```

---

```
public class MyProgram {  
    public static void main(String[] args) {  
        Student x = new Student();  
        x.name = "Fred Schnurd";    // OK!  
        // etc.
```



# Visibility, cont.

## - Private visibility:

```
public class Student {  
    public String name;  
    private String ssn;  
    // etc.
```

---

```
public class MyProgram {  
    public static void main(String[] args){  
        Student x = new Student();  
        x.ssn = "123-45-6789"; // NOT OK TO MODIFY!  
        System.out.println(x.ssn); // NOT OK TO ACCESS!
```

**"Variable ssn in class Student not accessible from class MyProgram"**

# Visibility, cont.

- The same applies to methods:

```
public class Student {  
    public String name;  
    private String ssn;  
    // etc.  
    public void someMethod() { ... }  
    private void anotherMethod() { ... }
```

---

```
public class MyProgram {  
    public static void main(String[] args){  
        Student x = new Student();  
        x.someMethod(); // OK  
        x.anotherMethod(); // NOT OK!
```

# Info. Hiding, cont.


(glass wall)

*Client code may only access features that are declared to be public (typically, method signatures).*

**PUBLIC:**

Method signatures:

registerForCourse() {  }

dropCourse() {  }

<etc.>

*The internal details of method code bodies are effectively private ...*

**PRIVATE:**

Attributes:

name

ssn

address

<etc.>

(brick walls)

*... as are most attributes.*

# Accessor/Modifier Methods

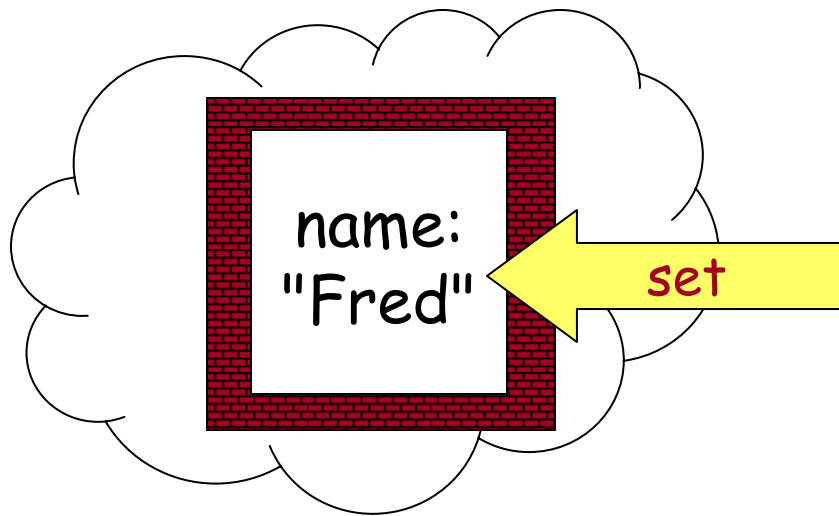
- If we make all of the attributes private, how will we ever get or set their values from the main program?

# Get/Set Methods

- If we make all of the attributes private, how will we ever get or set their values from the main program?
  - Via public accessor/modifier methods (aka "get" and "set" methods)

# Get/Set Methods

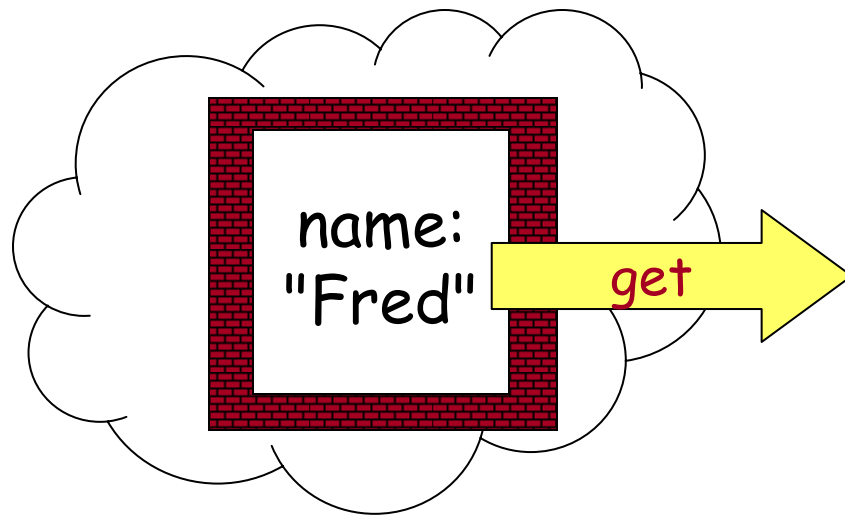
- If we make all of the attributes private, how will we ever get or set their values from the main program?
  - Via public accessor/modifier methods (aka "get" and "set" methods)



"set" method passes in a new value for a given attribute from client code ...

# Get/Set Methods, cont.

- If we make all of the attributes private, how will we ever get or set their values from the main program?
  - Via public accessor/modifier methods (aka "get" and "set" methods)



... and the corresponding "get" method retrieves its value for use by client code

# Get/Set Methods, cont.

- For an attribute declaration of the form:

*visibility attribute-type attributeName;*

private String ssn;

- 'get' method syntax:

*public attribute-type getAttributeName()*

public String getSsn()

- 'set' method syntax:

*public void setAttributeName(attribute-type argument-name)*

public void setSsn(String newSsn)



# Interactive Exercise

- Write the get/set method signatures for the following attributes:

```
public class InventoryItem {  
    private double price;  
    // etc.  
}
```

---

---

```
public class LawFirm {  
    private Office location;  
    // etc.  
}
```

# Interactive Exercise, cont.

```
public class Department {  
    private Professor chairman;  
    // etc.  
}
```

# Standard Get/Set Methods

- 'Get' and 'set' method bodies are often simple 'one-liners':

```
public class Student {  
    private String name;  
    private String ssn;  
    // etc.  
  
    public String getSsn() {  
        return ssn;  
    }  
  
    public void setSsn(String newSsn) {  
        ssn = newSsn;  
    }  
    // etc.
```

# Invoking Get/Set Methods

- We invoke get/set methods from client code via dot notation, just like any other method:

```
public class MyApp {  
    public static void main(String[] args) {  
        Student s = new Student();  
        s.setName("Fred Schnurd");  
        s.setSsn("123-45-6789");  
        s.setMajor("MATH");  
        // etc.  
        // or:  
        System.out.println(s.getName());  
        if (s.getAge() >= 21) { do something ... }  
        // etc.  
    }  
}
```

11/11/2009  
}

# JavaBeans

- Adhering to this syntax enables us to create highly reusable classes known as JavaBeans (not to be confused with Enterprise JavaBeans (EJBs))
- A JavaBean is simply a class that has been written in such a way that its "properties" can be discovered by a web server or an enterprise application server
  - Such applications know how to "talk to" a class to ask it for its structure
  - A property is represented by a get/set method pair

# JavaServer Pages and JavaBeans

- Can use JSP tags to access such properties:

```
<HTML>
<HEAD>
<TITLE>Banking Customer Information</TITLE>
</HEAD>
<BODY>
    Equivalent to: Person person = new Person();
    <!-- Retrieve the Person object from the request. -->
    <jsp:usebean id="person" scope="request" class="Person" />
    Customer Name:  <jsp:getProperty name="person" property="name" /> <BR>
    Pin No.:  <jsp:getProperty name="person" property="pinNo" /><BR>
</BODY>
</HTML>
    Equivalent to: person.getName();
```

## JavaBeans, cont.

- The significance of JavaBeans with respect to web deployment of Java applications is covered in the sequel to this course, "Deploying Java Objects"

# Return Types

- Note that a method can only return one result or answer - a simple data type or a single object reference

```
public int getAge() { ... }  
public Professor getAdvisor() { ... }
```

- This result can be an object of arbitrary complexity, including a special type of object called a **collection** that can contain multiple other objects - we'll learn about this later



# Encapsulation, Revisited

- It is useful to think of an object as a 'fortress' that 'guards' its data
  - We cannot directly access the values of an object's *privately*-declared attributes without an object's permission and knowledge, i.e. ...
  - We must use one of an object's *publicly* accessible methods
    - Example: wallet access
- An object is responsible for ensuring the integrity of its own data

# Info. Hiding Helps to Ensure Data Integrity

- As an example, let's say that we wish to store a Student's birthdate as a String attribute:

```
private String birthdate;
```

- Our intention is to record birthdates in the format "mm/dd/yyyy"
- We provide a Student method with signature:

```
boolean updateBirthdate(String d)
```

- The single argument to the updateBirthdate() method as we've declared it is a String which can be of literally any format
- How will we ensure that dates conform to our desired format?

## Info. Hiding, cont.

- We'll provide logic within the method which verifies and, if necessary, rejects the request

```
public class Student {  
    private String birthdate;  
    // other details omitted  
  
    public boolean updateBirthdate(String d) {  
        if (date not in the format mm/dd/yyyy) return false;  
        else (if mm not in the range 01 to 12) return false;  
        // etc. for other validation tests  
        else {  
            // All is well!  
            birthdate = d; // or even: setBirthdate(d);  
            return true;  
        }  
    }  
}
```

# Info. Hiding Simplifies Code Maintenance

- Info. hiding insulates our application from changes to the hidden implementation details of a class
  - As an example, let's say that we design our Student class to have an attribute:

```
private int age;
```

- and a get method as follows:

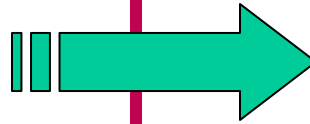
```
public int getAge() {  
    return age;  
}
```

## Info. Hiding, cont.

- We test and deploy our Student class, and it is used in dozens of applications throughout the University
- Later on, we decide that it was a silly idea to implement an age attribute -- we should have instead implemented a birthdate attribute, and compute the age whenever it is needed
- We change our class design as follows (see next slide):

# Info. Hiding, cont.

```
public class Student {  
    // Explicit age attribute.  
    private int age;  
  
    public int getAge() {  
        return age;  
    }  
  
    // etc.
```



```
public class Student {  
    // NO age attribute!  
    private String birthDate;  
  
    public int getAge() {  
        return systemDate -  
            birthDate;  
    }  
  
    // etc.
```

- The beauty is that, because all we've changed are private details of the class, *we don't care which of these approaches is used!* (cont.)

## Info. Hiding, cont.

- With EITHER version of the Student class, the public method signature for `getAge()` is the same, and so client code won't have to change:

```
Student s = new Student();  
// details omitted ...  
int i = s.getAge();
```

- This method call worked before, when "age" was an explicit attribute; and it works now, when age is computed from the birthdate attribute
- Code changes made only to the private aspects of a class are said to be **encapsulated**

# Info. Hiding, cont.

- Of course, all bets are off if the developer of a class changes one of its public method signatures
- For example, if we decided to change

`int getAge()`

to:

`float getAge()`

then the client code

`int i = s.getAge();`

would no longer compile, and we would have a "ripple effect" throughout our application(s)



# Info. Hiding, cont.

- Of course, all bets are off if the developer of a class changes one of its public method signatures
- For example, if we decided to change

`int getAge()`

to:

`float getAge()`

then the client code

`int i = (int) s.getAge(); // a cast is needed`

would no longer compile, and we would have a "ripple effect" throughout our application(s)

# Accessing One's Own Features

- Whether public or private, a class's features are visible/accessible within all of its own methods
  - This is because the attributes are "in scope" at the class level

```
public class Student {  
    private String name;  
  
    public void printInfo() {  
        System.out.println(name);  
        // etc.  
    }  
  
    // etc.
```

# Accessing One's Own Features, cont.

- We can also invoke one method of a class from inside another method of the same class without using dot notation
  - The object receiving the message is implied to be the object whose method is being executed

```
public class Student {  
    public void printTranscript(String fileName) {  
        details omitted ...  
    }  
  
    public void printInfo(String fileName) {  
        printTranscript(fileName);  
        do some other stuff ...  
    }  
}
```

# Initialization, Revisited

- We learned earlier that when primitive (i.e. non-object) variables are declared, their values are not automatically initialized
- Trying to access such variables without explicitly initializing them will result in a compilation error

```
public static void main(String[] args) {  
    // Declare several local variables.  
    int i;    // not automatically initialized  
    int j;    // ditto  
    j = i;    // compilation error!
```

Variable i might not have been initialized.

# Initialization, Revisited

- This is also true if we try to access the value of a reference variable that is declared locally to a method but hasn't been explicitly initialized:

```
public static void main(String[] args) {  
    // Declare a local reference variable.  
    Student s;  
  
    // A common way to test whether a reference  
    // variable is holding onto an object;  
    // however, this would result in ...  
    if (s == null) { ... } // ... a compilation error!
```

Variable *s* might not have been initialized.

# Initialization, Revisited

- To avoid such compilation errors, always initialize locally (method-) scoped variables:

```
public static void main(String[] args) {  
    // Locally scoped variables.  
    int i = 0;  
    Student s = null; // Explicit initialization.  
    // or:  
    Student s2 = new Student(); // Also explicit.  
  
    // A common way to test whether a reference  
    // variable is holding onto an object.  
    if (s == null) { ... } // This will now compile.
```

# Initialization, cont.

```
public class SomeClass {  
    // Attributes.  
    String name;  
    // etc.  
  
    // Methods.  
    public void someMethod() {  
        // These variables are local to this method;  
        // initialize them explicitly.  
        double x = 0.0;  
        String s = null;  
        // or:  
        String t = "foo";  
        // etc.  
    }  
}
```

# Initialization, cont.

- However, with *attributes*, which are declared at the class scope level:

```
public class Student {  
    int age;  
    boolean isHonorsStudent;  
    Professor advisor;  
    // etc.
```

they *are* automatically initialized to the appropriate zero-equivalent values in Java:

- booleans are initialized to false
  - numerics are initialized to either 0 or 0.0,
  - object references are initialized to `null`
- and so forth



# Exercise #4

# Functional Decomposition, Revisited ...

- With functional decomposition, data was an "afterthought"
  - Was passed around from one function to the next, like a car on an assembly line
  - Data structure had to be understood in *MANY* places throughout an application
  - If the data structure changed, there were major "ripple effects"
  - If data got corrupted, it was hard to pinpoint where (in what function) this had occurred

## ... vs. the OO Approach

- With OOSD:
  - We consider the data aspects of a model first, functional aspects later
  - Data is encapsulated inside of objects
  - (Private) data structure only has to be understood by the object (class) it belongs to
  - If the (private) data structure changes - or, for that matter, if ANYTHING private changes
    - there are virtually NO "ripple effects"

(continued)

## ... vs. the OO Approach

- With OOSD (cont.):
  - An object is responsible for ensuring the integrity of its own data
  - If data becomes corrupted, we can pretty much assume it was the object itself that did the corrupting!
    - Look for flawed business logic within one or more methods

# Software at its Simplest, Revisited

- If every software application consists of:
  - Data
  - Functions that operate on the datathen ...

# Software at its Simplest, Revisited

- If every software application consists of:
  - Data
  - Functions that operate on the datathen ...
- An object can hence be thought of as a sort of "mini application"
  - Its methods (functions) operate on its attributes (data)

# Exceptions to the Public/Private Rule

- Even though it is *generally* true that:
  - Attributes are declared to be private,
  - Method signatures are declared to be public, and
  - Private attributes are teamed up with public get/set methods,there are numerous exceptions to this rule

# Exceptions to the Public/Private Rule, cont.

- Exception #1: If we provide only a 'get' method for an attribute, then that attribute is effectively read-only
- How are these attributes' values ever set?
  - When we first create the object, through use of a special type of function called a constructor
  - As a side effect of calling some other method
    - Assign a grade to a Student => gpa is recalculated



# Exceptions to the Public/Private Rule, cont.

- Exception #2: An attribute may be used strictly for internal housekeeping purposes --> neither a 'get' nor a 'set' method is provided

```
public class Student {  
    private int countDsAndFs;  
  
    public boolean onAcademicProbation() {  
        if (countDsAndFs > 1) return true;  
        else return false;  
    }  
  
    // etc.
```

# Exceptions to the Public/Private Rule, cont.

- Exception #3: Some methods may be used strictly for internal housekeeping, as well, in which case these may also be declared private

```
public class Student {  
    private Date birthdate;  
  
    public boolean over21() {  
        if (computeAge() > 21) return true;  
        else return false;  
    }  
  
    public int getAge() {  
        return computeAge();  
    }  
  
    private int computeAge() {  
        // Pseudocode.  
        return system date - birthdate;  
    }  
}
```

11/11/2009  
}

Copyright © 2005 Jacquie Barker

266

# Exceptions to the Public/Private Rule, cont.

- Exception #4: Classes may occasionally declare public attributes as a convenience when such attributes are going to be accessed frequently (e.g., the Point class)

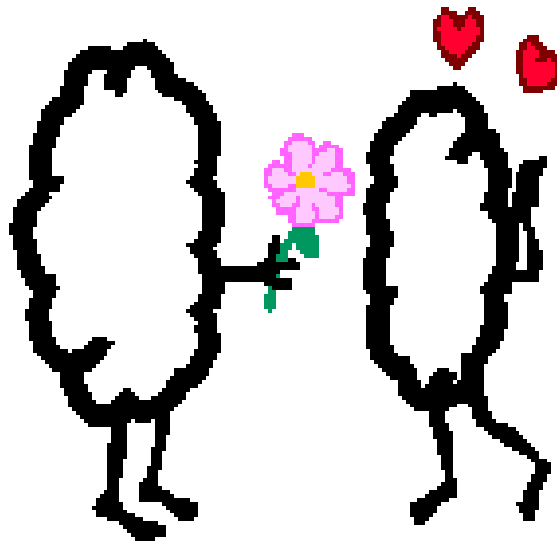
```
Point p = new Point();  
p.x = 7.3;  
p.y = 3.0;
```

# "Full Disclosure"

- There are actually two other types of visibility, known as:
  - protected visibility
  - default/package visibility
- We'll talk about these later in the course
- But, there is a lot more to learn about objects and Java first!

# Relationships Between Objects

(Chapter 5)



# Encapsulation

- Everything that an object needs in order to fulfill its mission in an application is, in theory, encapsulated within the object, either:
  - As an attribute (data) or
  - As a method (behavior)
- This phenomenon is known as **encapsulation**
- The reality, however, is that virtually no object can operate in isolation
- Objects must collaborate, sharing their data and behaviors, in order to collectively fulfill the "higher purpose" of an application
  - Like employees in a corporation
  - Like cells in our body

# Object Collaboration

- In order for objects to collaborate, they need to be able to find one another
- By maintaining object references as attributes, an object 'A' can be 'hardwired' to object 'B' in memory in order to collaborate with it
  - Like having someone's phone number

# Objects as Attrib., Revisited

```
// File: Student.java
```

```
public class Student {
```

```
    // Attributes.
```

```
    String name;
```

```
    String studentID;
```

```
    String birthdate;
```

```
    String address;
```

```
    String major;
```

```
    float gpa;
```

```
    Professor advisor;
```

```
    // ??? courseLoad;
```

```
    // ??? transcript;
```

```
    // Methods go here ... stay tuned! :o)
```

```
}
```

A Student object  
maintains a handle on its  
advisor (Professor)  
object as an attribute ...



# Objects as Attrib., cont.

```
// File: Professor.java
```

```
public class Professor {
```

```
    // Attributes.
```

```
    String name;
```

```
    String employeeID;
```

```
    Student advisee;
```

```
    double salary;
```

```
    // etc.
```

```
    // Methods.
```

```
    public double getSalary() {
```

```
        return salary;
```

```
    }
```

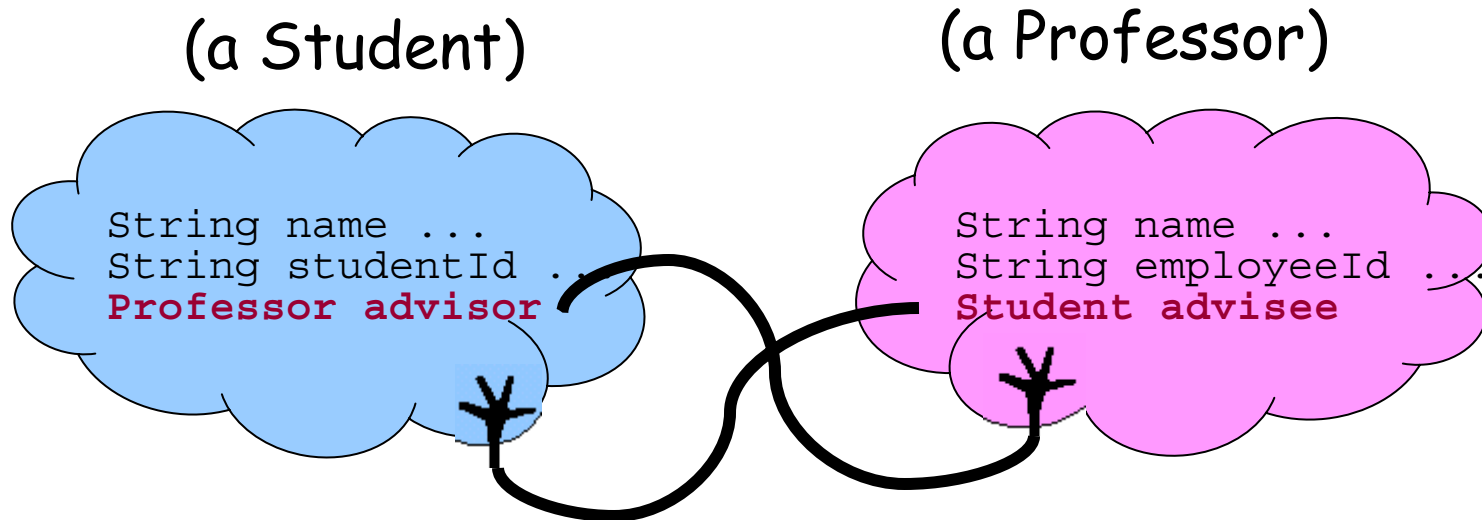
```
    // etc.
```

```
}
```

← ... and vice versa!

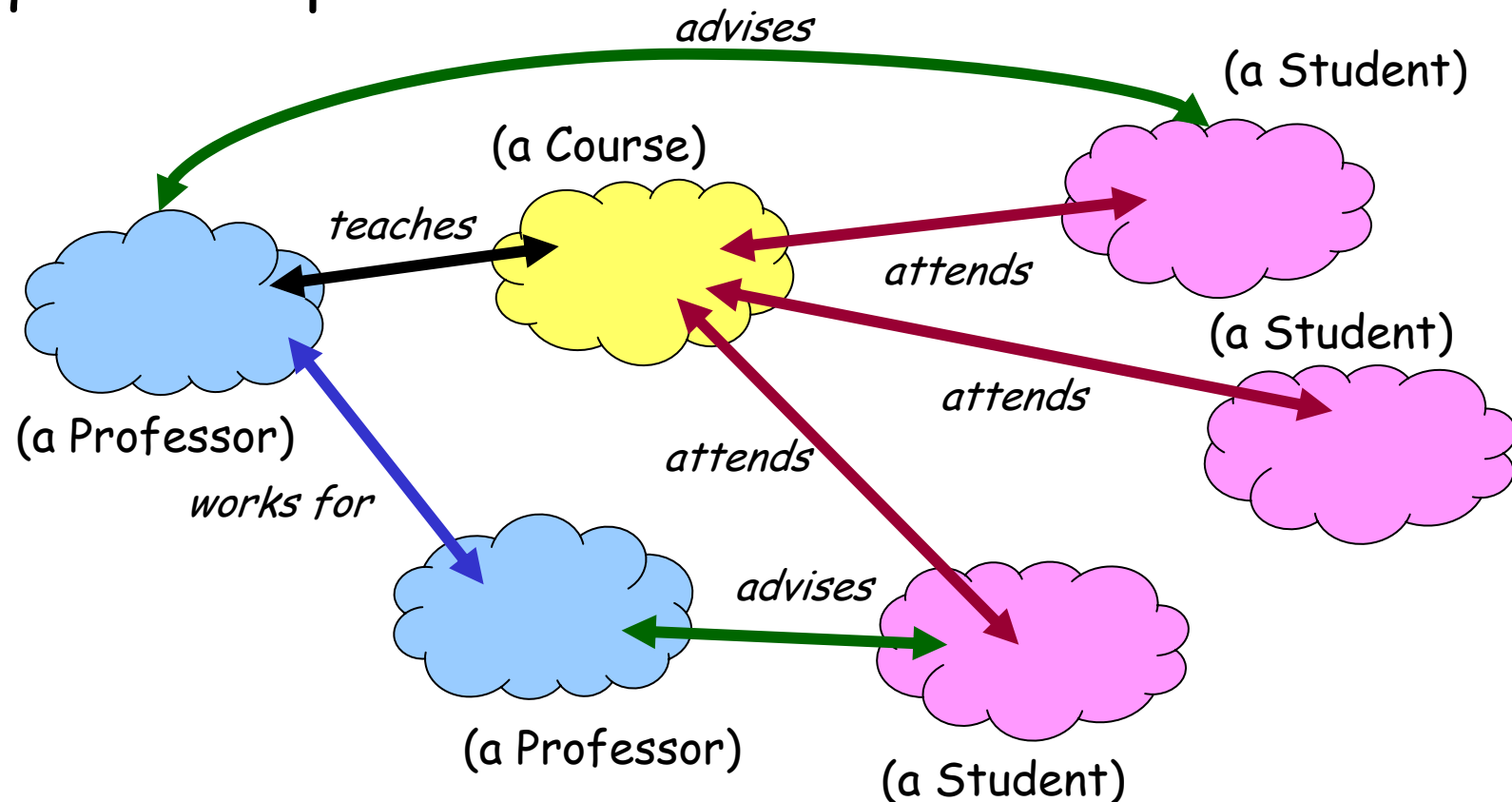
# Objects as Attributes, cont.

- As we discussed earlier, collaborating objects are thus linked in memory



# Objects as Attributes, cont.

- Collaboration networks among objects can be quite complex



# Objects as Attributes, cont.

- Part of the challenge of object modeling is to anticipate how objects are going to need to collaborate in carrying out the mission of the application
- We then build the *potential* for such connections into the data structure ("bone structure") of our classes (objects as attributes) so that we can "wire together" individual collaborating objects at run time

# Associations and Links

- The formal name for a structural relationship that exists between classes is an **association**
- Some sample associations:
  - A Student *is enrolled in* a Course
  - A Professor *teaches* a Course
  - A Degree Program *requires* a Course
- The term **link** is used to refer to a structural relationship that exists between two specific *objects/instances*

# Associations and Links, cont.

- Given the association a Student *is enrolled in* a Course', we might have the following links:
  - Joe Blow (a particular Student object) is enrolled in Math 101 (a particular Course object)
  - Fred Schnurd (a particular Student object) is enrolled in Basketweaving 972 (a particular Course object)
  - Mary Smith (a particular Student object) is enrolled in Basketweaving 972 (a particular Course object; as it turns out, the *same* Course object that Fred Schnurd is linked to)

# Associations and Links, cont.

- In the same way that an object is a specific instance of a class, a link is a specific instance of an association with its member objects filled in

Association: ..... is enrolled in .....  
(some Student) (some Course)

Link: James Conroy is enrolled in Phys.Ed 311.  
(a specific Student) (a specific Course)

- Another way to think of the difference:
  - An association is a *potential* relationship between objects of a certain type/class, whereas
  - A link is an *actual* relationship between objects of those particular types

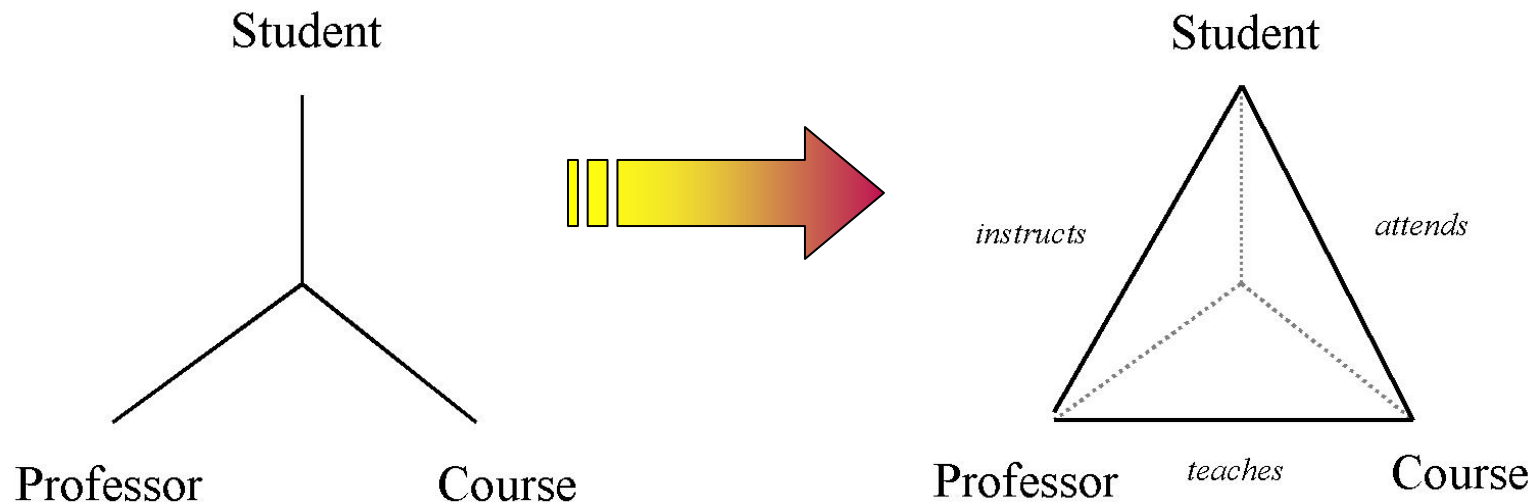
# Associations and Links, cont.

- Binary association: between two classes
- Reflexive association: between two instances of the same class: "A Course is a prereq. for a (different) Course"



# Associations and Links, cont.

- Higher order associations are possible, but rare
- We usually decompose these into multiple binary associations (eliminating the least interesting of these)



# Multiplicity

- For a given association type X between classes A and B, the term **multiplicity** refers to the number of objects of type A that may be linked with a given instance of type B
  - For example, a Student attends multiple Courses, but a Student has only one Professor in the role of advisor

# Multiplicity, cont.

- Three cases:
  - One-to-one: "A Professor chairs exactly one Department, and a Department has exactly one Professor in the role of chairperson."
  - One-to-many: "A Department employs many Professors, but a Professor (usually) works for exactly one Department."
  - Many-to-many: "A Student attends many Courses, and a Course has many Students in attendance."

# Multiplicity, cont.

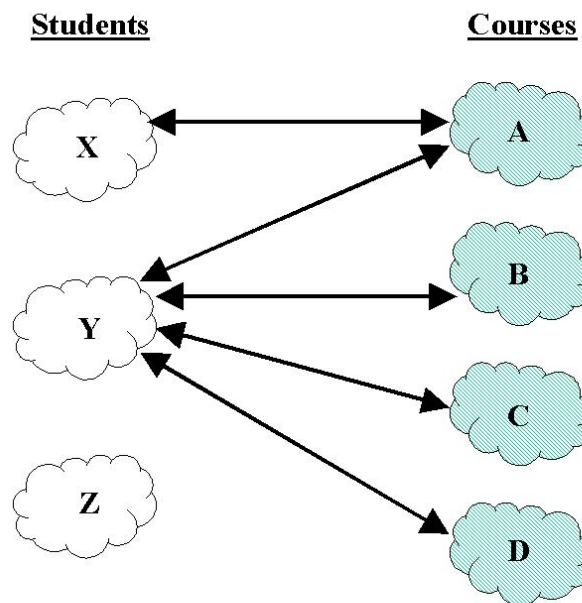
- Optional versus mandatory participation:
  - One-to-one: "A Professor *optionally* chairs exactly one Department, *but it is mandatory that* a Department has exactly one Professor in the role of chairperson."
  - 'Many' in one-to-many and many-to-many can be interpreted as either 'zero or more (optional)' or as 'one or more (mandatory)':
    - A Department employs **one or more** ('many'; *mandatory*) Professors, ...
    - A Professor advises **zero or more** ('many'; *optional*) Students, ...

# Multiplicity and Links

- Note that the concept of multiplicity pertains to associations, but not to links
- **Links always exist in pairwise fashion between two objects** (or, in rare cases, between an object and itself)
- So, multiplicity in essence defines how many links of a certain association type can originate from a given object => let's look at a few examples

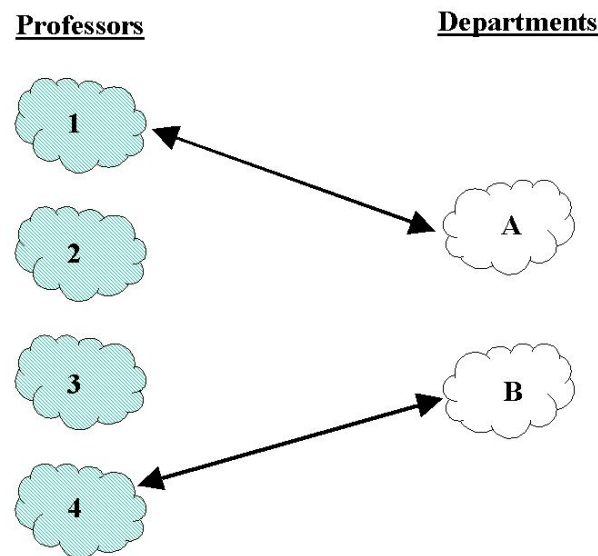
# Multiplicity and Links, cont.

- Many-to-many: 'A Student enrolls in zero or more Courses, and a Course has one or more Students enrolled in it.'

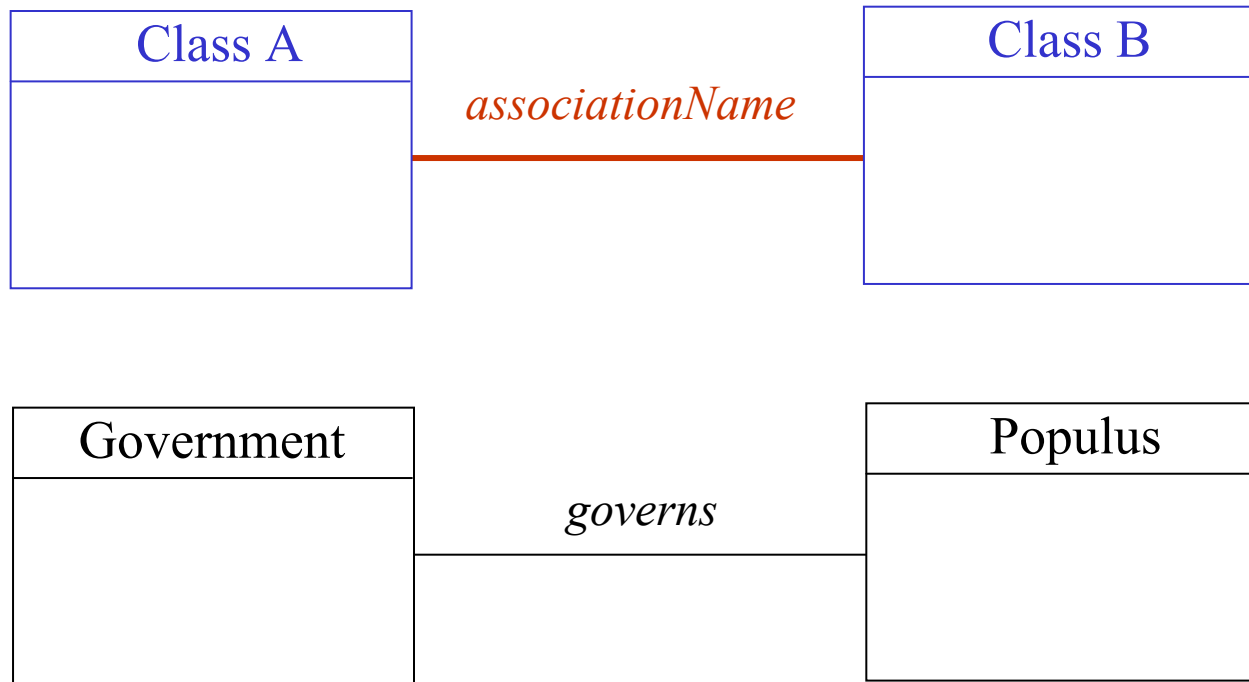


# Multiplicity and Links

- One-to-one: 'A Professor *optionally* chairs exactly one Department, and a Department has exactly one Professor in the role of chairman.'

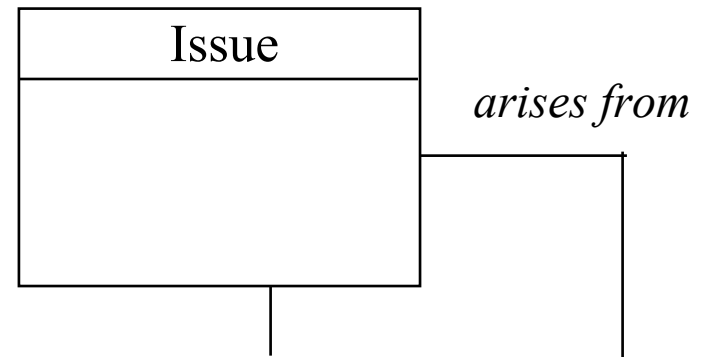
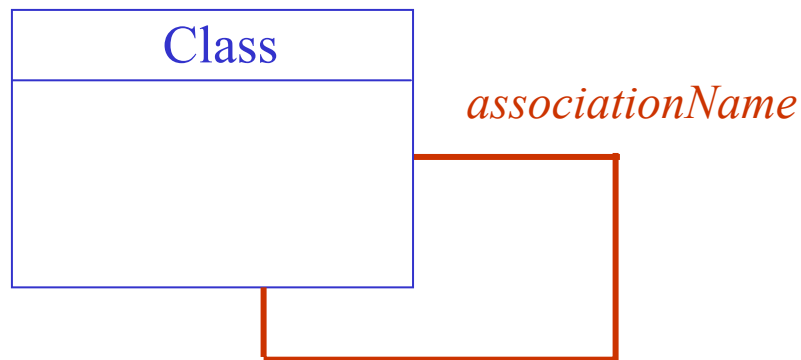


# Representing Associations in UML Notation

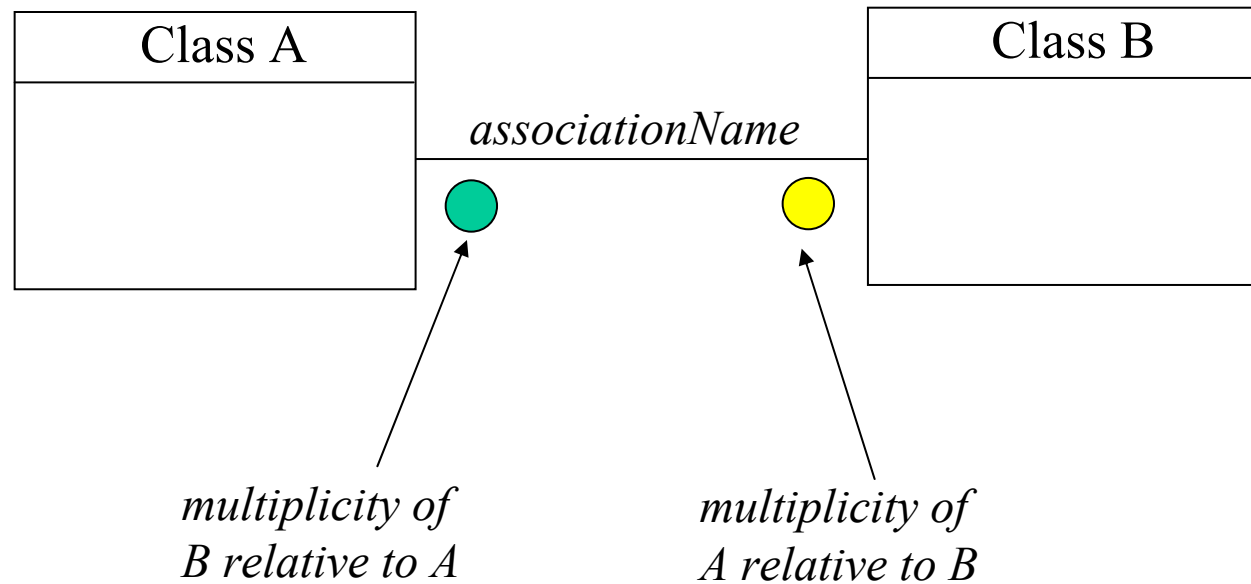




# Representing Reflexive Associations in UML Notation



# Representing Multiplicity in UML Notation



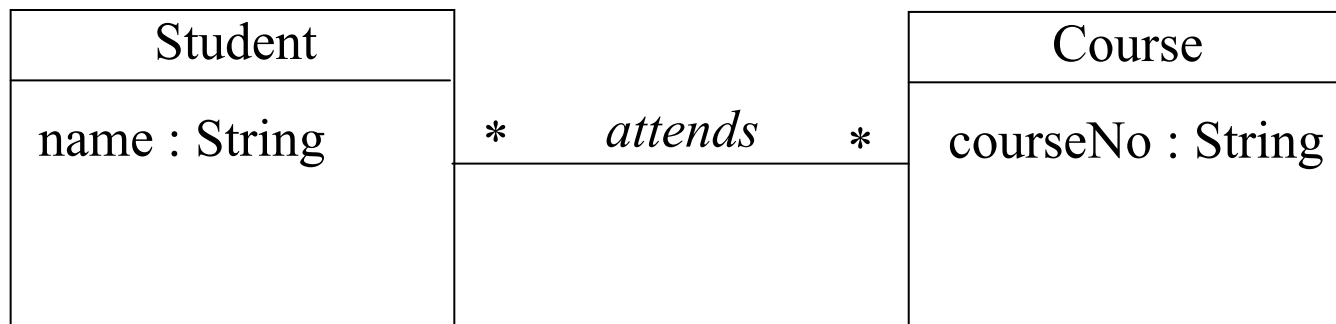
# Representing Multiplicity in UML Notation

- Exactly One: 1
  - One or More: 1..\*
  - Zero or More: 0..\*
  - Zero or One: 0..1
  - Many: \*
  - Specific number(s): 4..6
- 3
- 1..4

# Association Classes

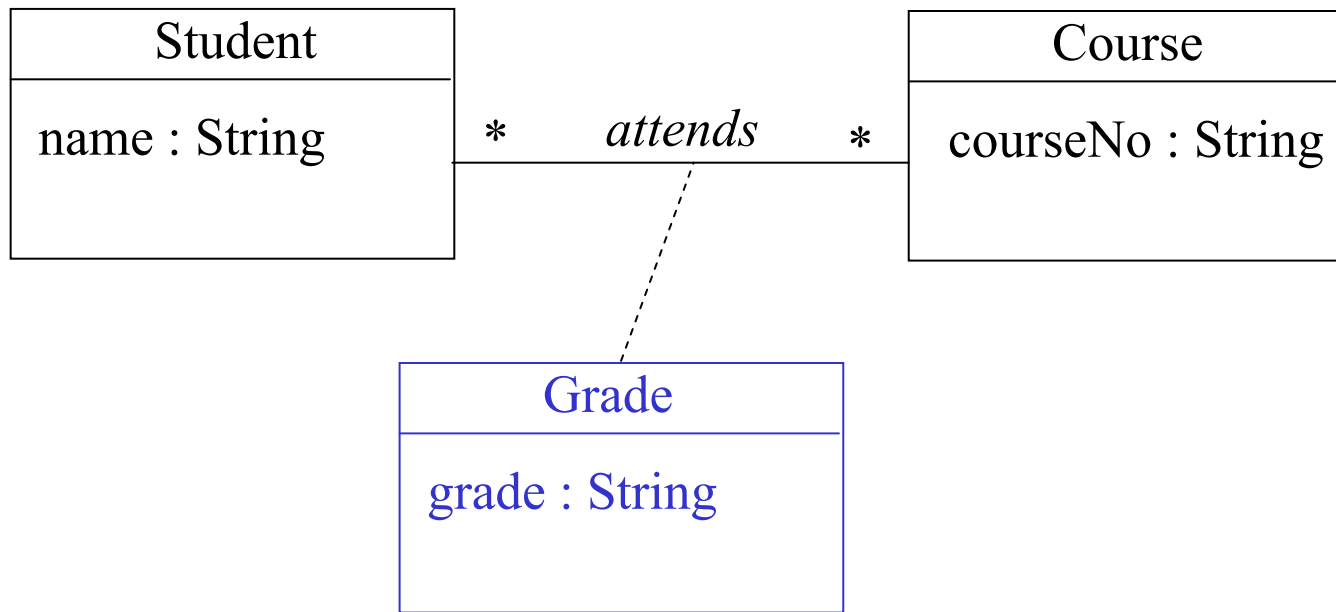
- On occasion, we have a need to model information that is associated with a link between objects rather than with an individual object
  - Example: the grade that is assigned to a student for a particular course

# Association Classes, cont.



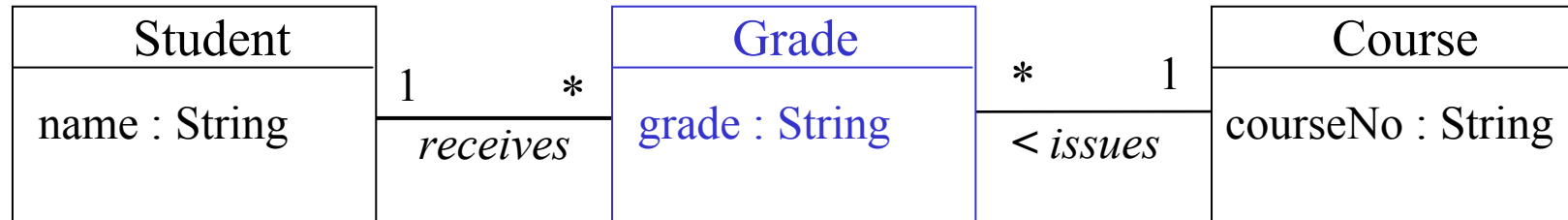
Where should the grade be reflected?

# Association Classes as UML



A grade is associated with each *link* between a Student and a Course

# Equivalent Representation



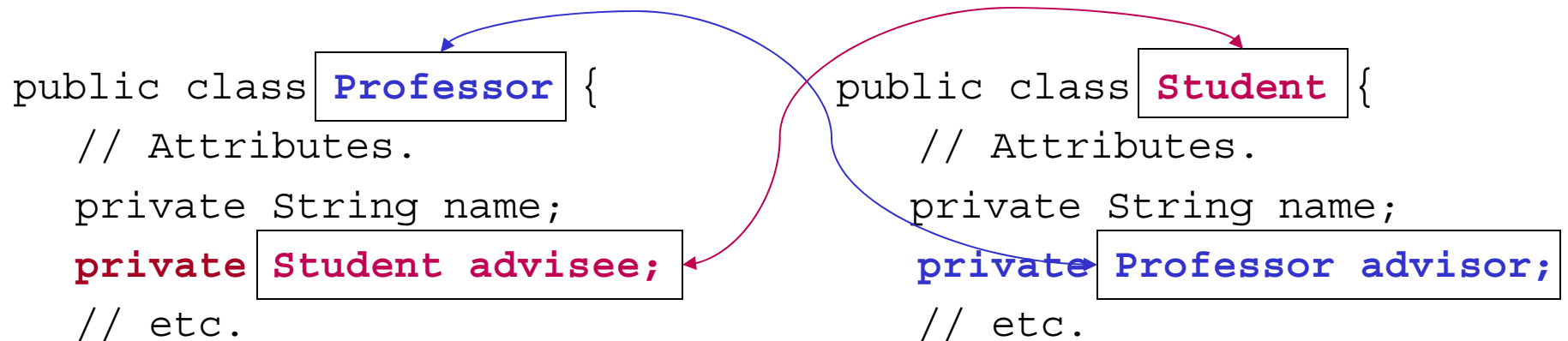
# Representing Associations In Code

- We represent associations in code through the use of object references as attributes
  - We use individual reference variables to represent the "one" end of an association
  - We use collections (which we'll talk about a bit later on) to represent the "many" end of an association



# Representing Associations In Code -- An Example

- A Professor advises optionally one Student, and a Student has exactly one Professor as an advisor



- We've seen this before! Each class holds a reference to the other as an attribute

# Associations in Code, cont.

- The get/set methods look like this:

```
public class Student {  
    private Professor advisor;  
    // etc.  
    public void setAdvisor(Professor a) {  
        advisor = a;  
    }  
    public Professor getAdvisor() {  
        return advisor;  
    }  
}
```

---

```
public class Professor {  
    private Student advisee;  
    // etc.  
    public void setAdvisee(Student a) {  
        advisee = a;  
    }  
    public Student getAdvisee() {  
        return advisee;  
    }  
}
```

# Get/Set Syntax Review

- These are standard get/set methods!
  - We've seen this syntax before: for an attribute:

*visibility attribute-type attributeName;*  
private Professor advisor;

- 'get' method syntax:

*public attribute-type getAttributeName()*  
public Professor getAdvisor()

- 'set' method syntax:

*public void setAttributeName(attribute-type argument-name)*  
public void setAdvisor(Professor newAdvisor)

# Associations in Code, cont.

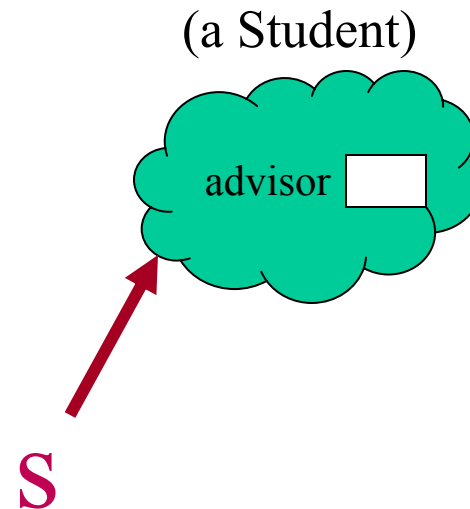
- The client code needed to bidirectionally link objects at run time would thus be as follows:

```
public static void main(String[] args) {  
    Student s = new Student();  
    Professor p = new Professor();  
    // details omitted ...  
    // Call the reciprocal "set" methods.  
    s.setAdvisor(p);  
    p.setAdvisee(s);  
}
```

# Associations in Code, cont.

- Memory Schematic:

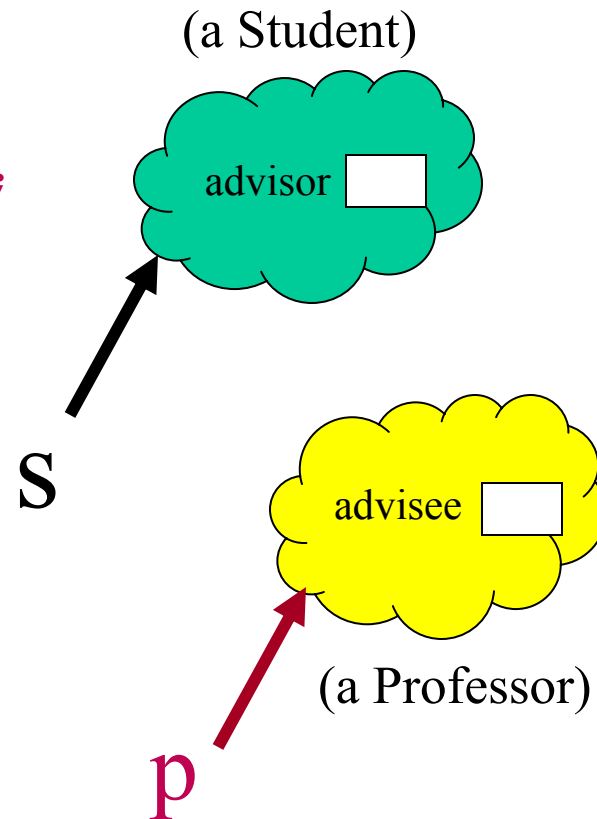
```
Student s = new Student();
```



# Associations in Code, cont.

- Memory Schematic:

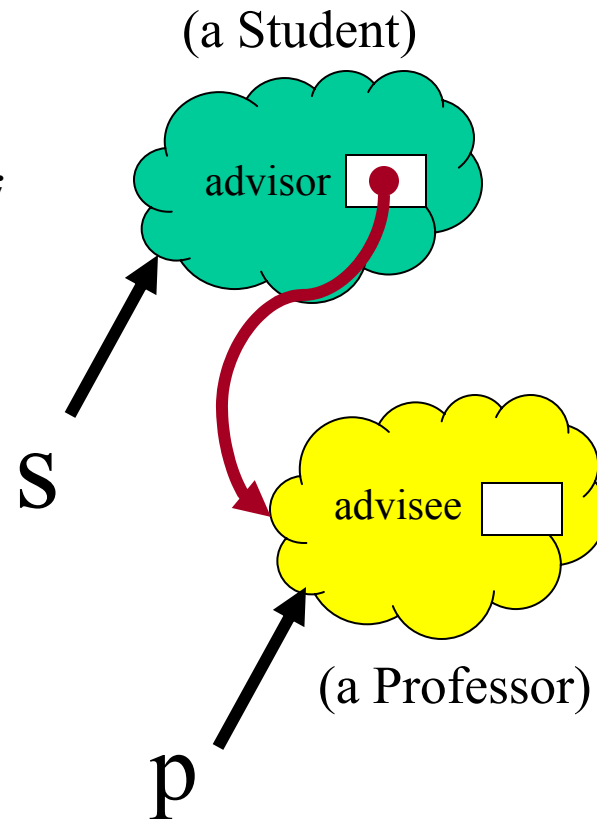
```
Student s = new Student();  
Professor p = new Professor();
```



# Associations in Code, cont.

- Memory Schematic:

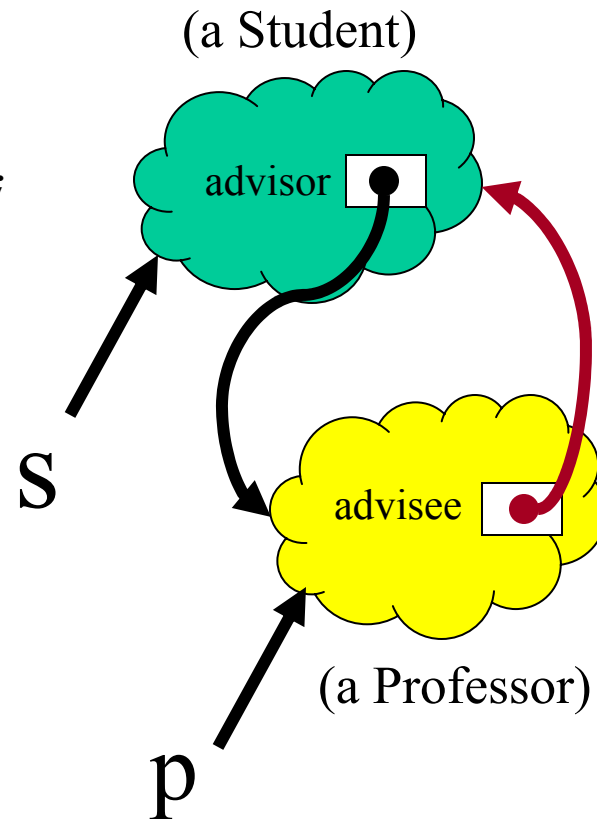
```
Student s = new Student();  
Professor p = new Professor();  
s.setAdvisor(p);
```



# Associations in Code, cont.

- Memory Schematic:

```
Student s = new Student();  
Professor p = new Professor();  
s.setAdvisor(p);  
p.setAdvisee(s);  
(memory sketch)
```





# Object References as Attributes

- What do we gain by defining the Student's `advisor` attribute as a reference to a Professor object vs. merely as a String?
  - We avoid data redundancy and the potential for loss of data integrity
    - E.g, if the Professor object's name changes for some reason, we don't want to have to hunt it down in the dozens/hundreds of Student objects for whom this professor is an advisor
  - By maintaining a handle on the Professor object, the Student object can also request *other services* of this Professor object (see next slide)

# An Example

```
public class Student {
    private String name;
    private String ssn;
    private Professor advisor;    // maintain a handle on a
    // etc.                      // Professor object

    public void setAdvisor(Professor a) {
        advisor = a;
    }

    public Professor getAdvisor() {
        return advisor;
    }

    public void display() {
        System.out.println("Student's name:  " + name);
        System.out.println("ID no.:  " + ssn);
        System.out.println("Advisor's name:  " +
            advisor.getName(); // request a service
        System.out.println("Advisor's Dept.:  " +
            advisor.getDept(); // request another service
    }
}
```

# Delegation

- If a request is made of an object 'A' and, in fulfilling the request, A in turn requests aid from another object 'B', this is known as **delegation by A to B**
  - Methods involving delegation are yet another example of **client code**

```
public class Student {
    Professor advisor;

    // This method is an example of client code:
    // a Student is a client of a Professor.
    public void display() {
        System.out.println("Student's name: " + name);
        System.out.println("ID no.: " + ssno);
        System.out.println("Advisor's name: " +
            advisor.getName()); // request a service
        System.out.println("Advisor's Dept.: " +
            advisor.getDept()); // request another service
    }
}
```

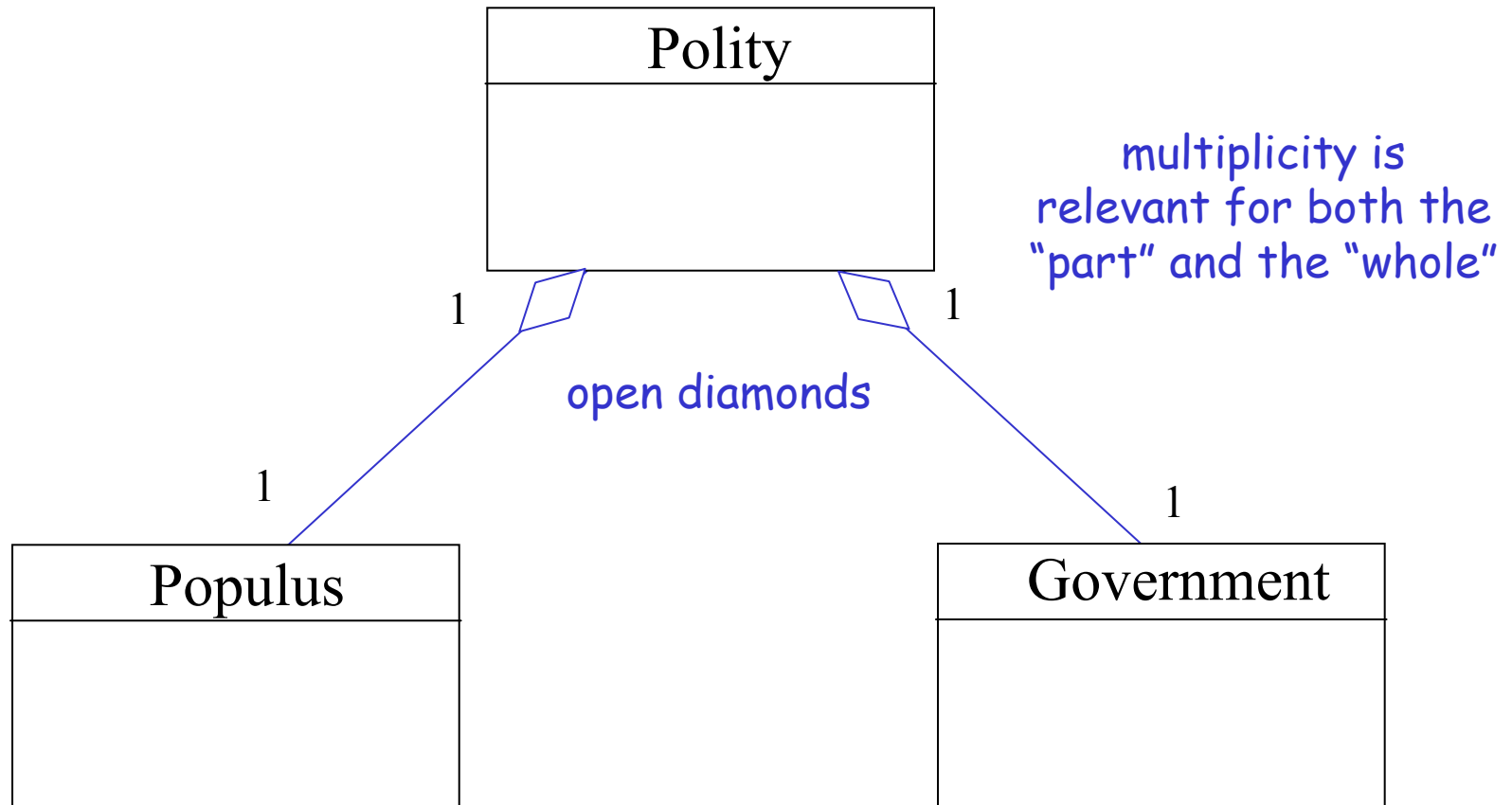
# Java Expressions, Revisited

- When we defined Java expressions earlier, there was one form that we omitted: messages; an expression in Java can thus be:
  - A constant: 7
  - A String literal: "foo"
  - A variable declared to be of either a simple or abstract data type: myString, x, facultyAdvisor
  - A message: z.length()
  - A 'chain' of 2 or more messages, concatenated by dots (.): aStudent.getAdvisor().getName().length()
  - Any two of the above that are combined with one of the Java binary operators: z.length() + 2
  - Any one of the above that is modified by one of the Java unary operators (++ , -- , ! , etc.): q++

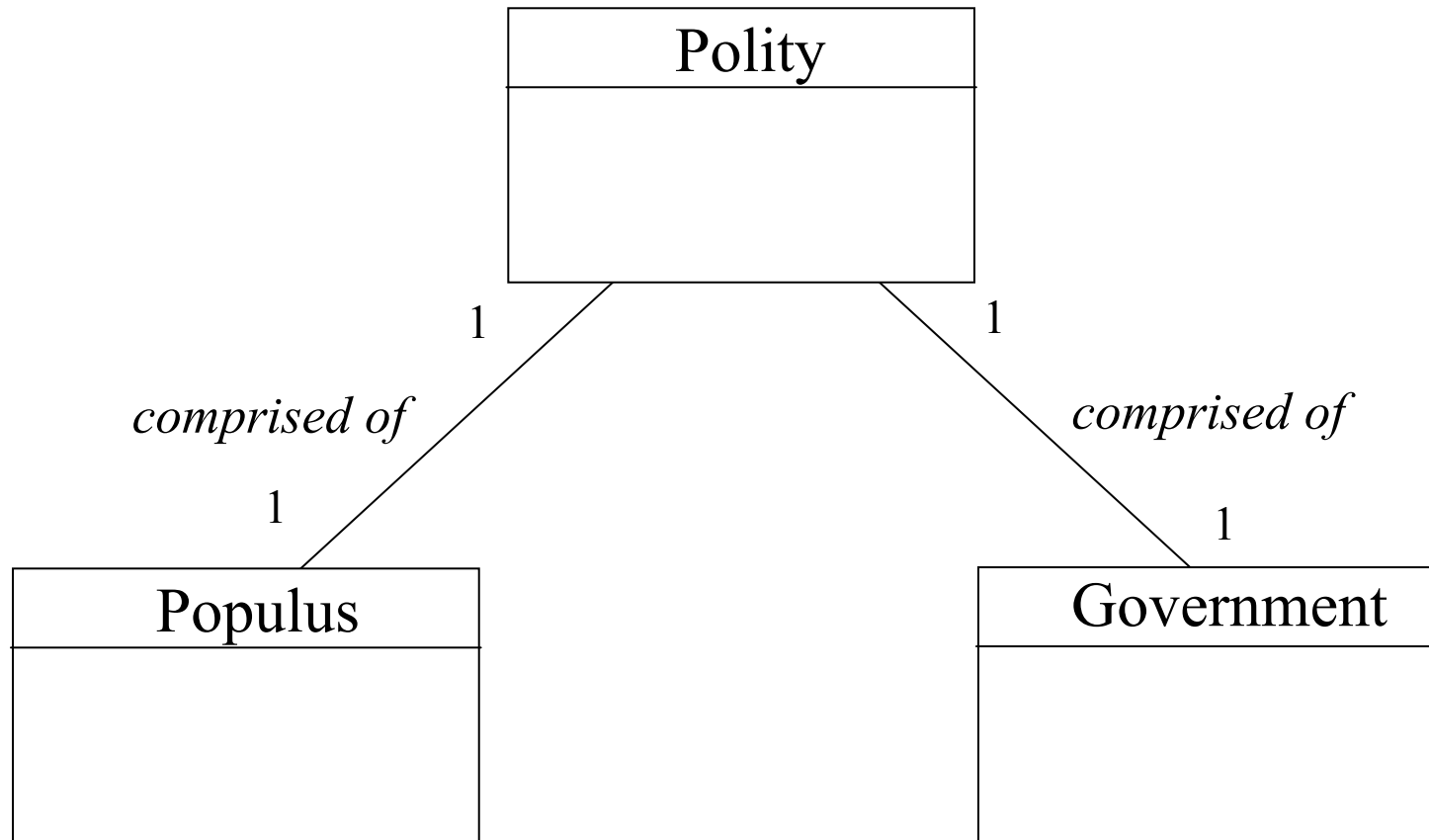
# Aggregation

- Special form of association that implies a "whole/part" or "is comprised of" type of relationship
- We might, for example, say:
  - A Faculty is comprised of Professors
  - A Polity is comprised of a Populus and a Government

# Aggregation as UML



# Equivalent UML

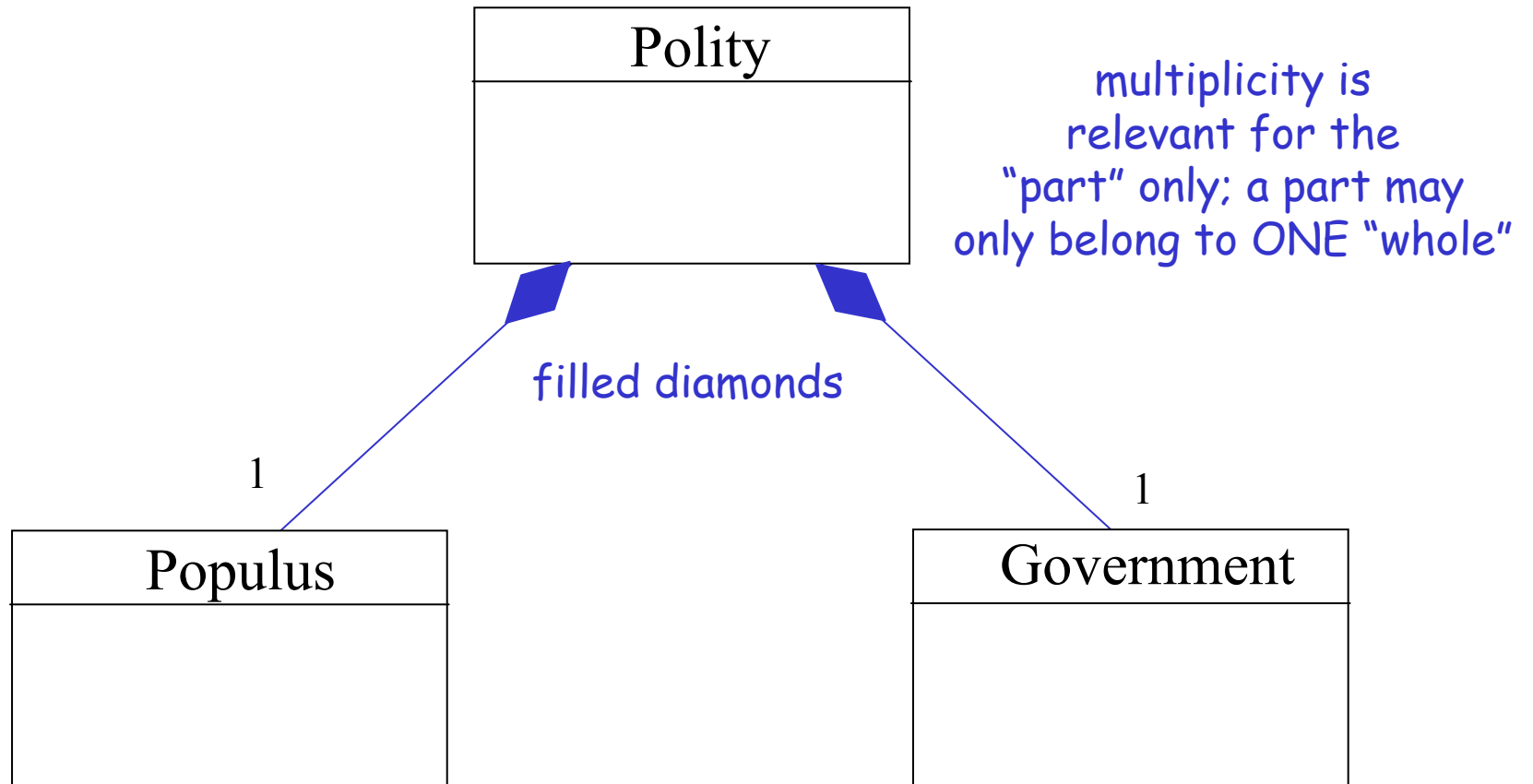


# Composition

- A “strong” form of aggregation, in which the “part” has a coincident lifetime with the “whole”
  - If the “whole” ceases to exist, so too do the “parts”
  - Example: A Building is comprised of Floors



# Composition as UML



# Exercise #5

# Exercise #5 Review

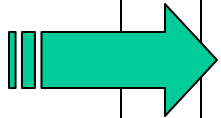
- Understanding object referencing is the “knothole” that we must get through in learning how to program in Java
  - Akin to learning pointers in C/C++
- Let's review what is happening in memory when we run the BankingApp that we developed for Exercise 5

JVM      java \_\_\_\_\_



JVM

java BankingApp



```
public class BankingApp {  
    public static void main(...) {  
        Person p = new Person();  
        BankAccount b =  
            new BankAccount();  
        p.setBankAccount(b);  
        b.setOwner(p);  
    }  
}
```

# JVM

## java BankingApp

```
public class BankingApp {  
    public static void main(...) {  
        Person p = new Person();  
        BankAccount b =  
            new BankAccount();  
        p.setBankAccount(b);  
        b.setOwner(p);  
    }  
}
```

```
public class Person {  
    String name;  
    BankAccount bankAccount;  
  
    // Methods go here ...  
}
```



# JVM java BankingApp

```
public class BankingApp {  
    public static void main(...) {  
        Person p = new Person();  
        BankAccount b =  
            new BankAccount();  
        p.setBankAccount(b);  
        b.setOwner(p);  
    }  
}
```

```
public class Person {  
    String name;  
    BankAccount bankAccount;  
  
    // Methods go here ...  
}
```



name  
bankAccount

# JVM java BankingApp

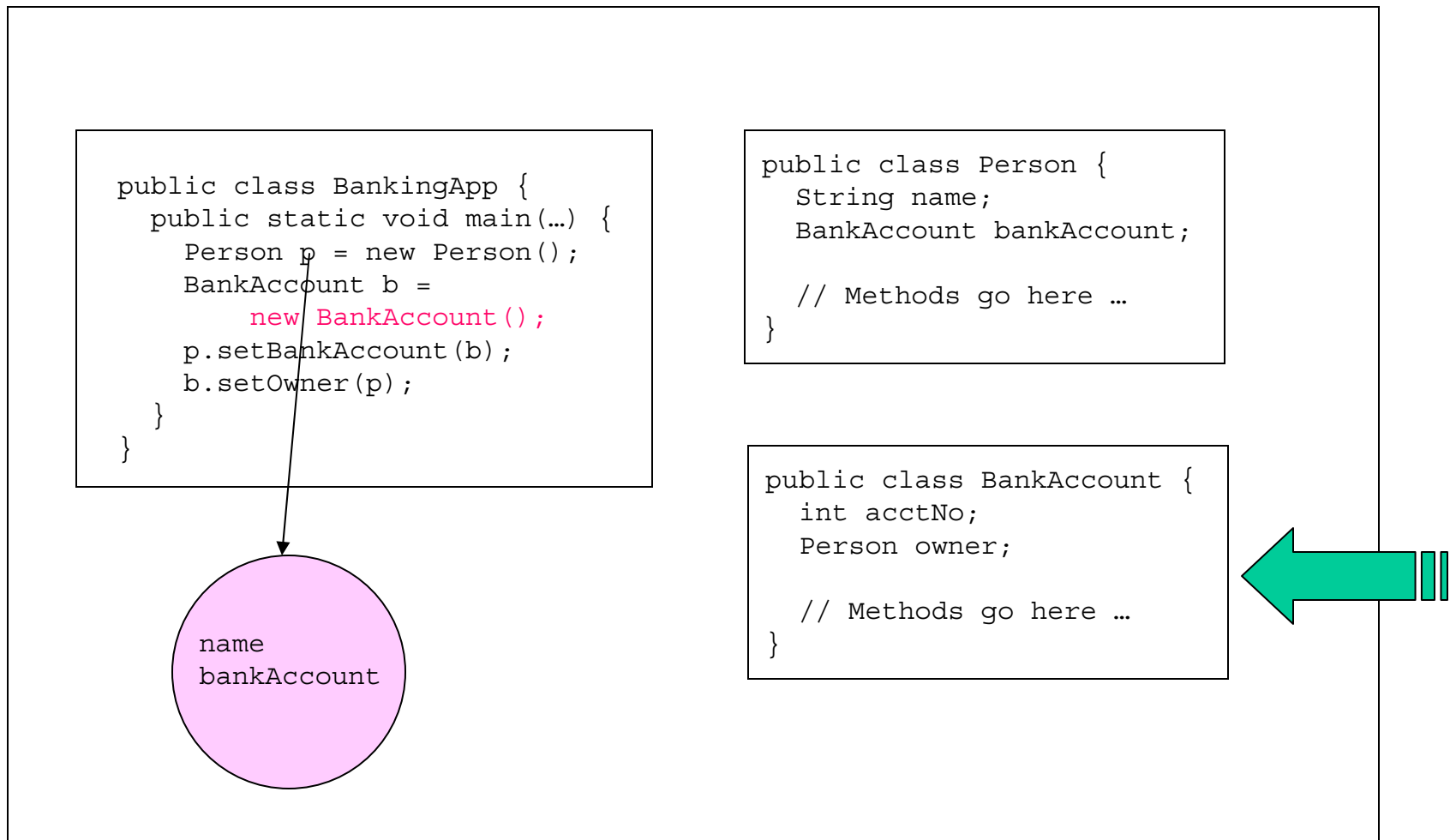
```
public class BankingApp {  
    public static void main(...) {  
        Person p = new Person();  
        BankAccount b =  
            new BankAccount();  
        p.setBankAccount(b);  
        b.setOwner(p);  
    }  
}
```

```
public class Person {  
    String name;  
    BankAccount bankAccount;  
  
    // Methods go here ...  
}
```

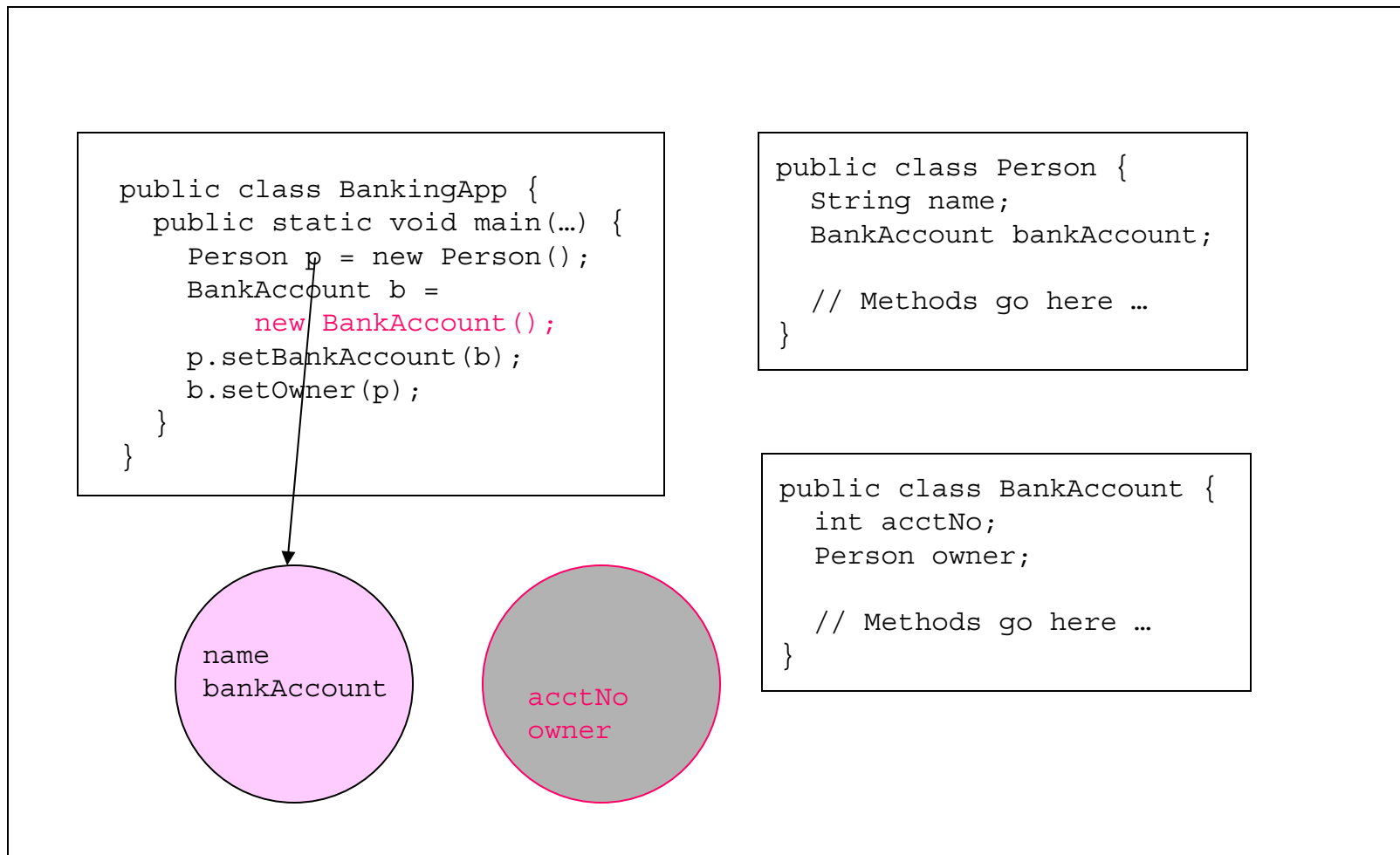




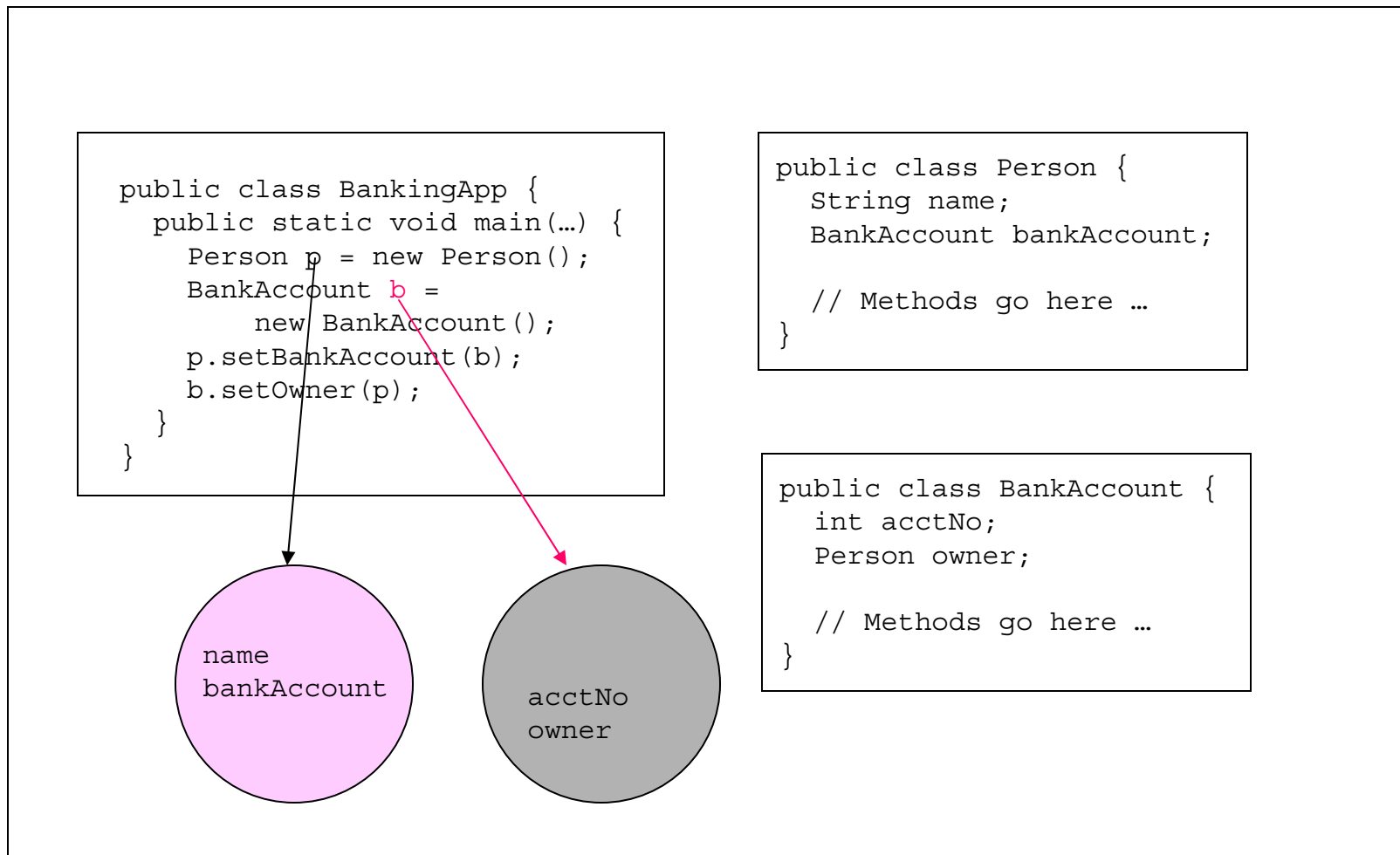
# JVM java BankingApp



# JVM java BankingApp



# JVM java BankingApp

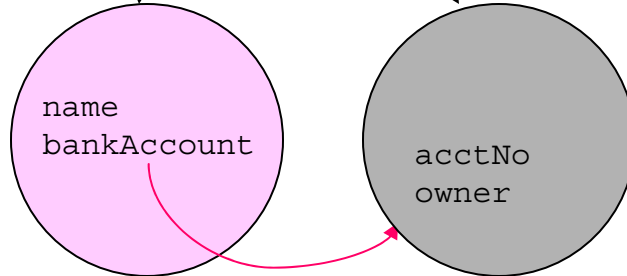


# JVM java BankingApp

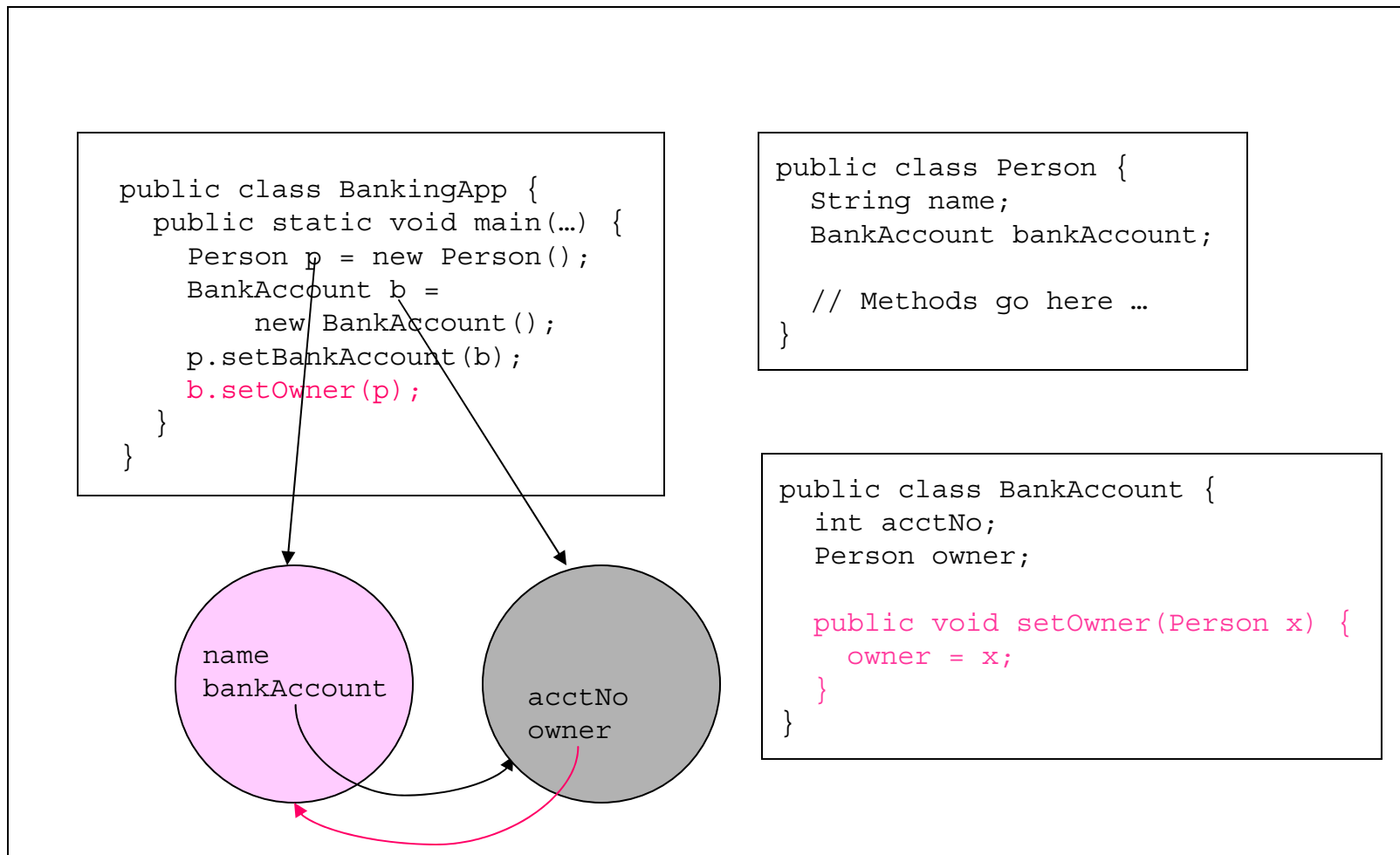
```
public class BankingApp {  
    public static void main(...) {  
        Person p = new Person();  
        BankAccount b =  
            new BankAccount();  
        p.setBankAccount(b);  
        b.setOwner(p);  
    }  
}
```

```
public class Person {  
    String name;  
    BankAccount bankAccount;  
  
    public void setBankAccount(BankAccount a) {  
        bankAccount = a;  
    }  
}
```

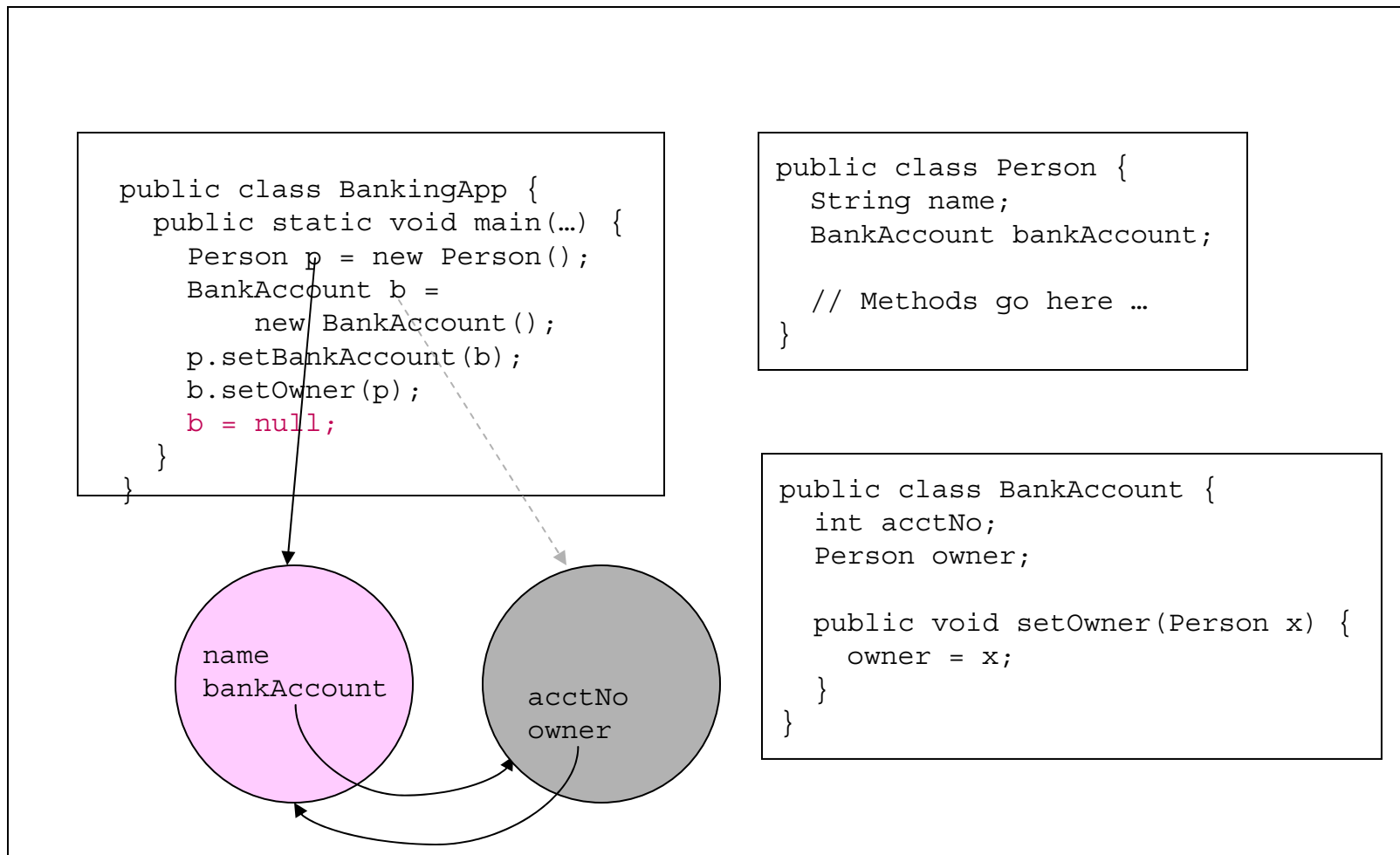
```
public class BankAccount {  
    int acctNo;  
    Person owner;  
  
    // Methods go here ...  
}
```



# JVM java BankingApp



# JVM java BankingApp



# Inheritance

- A powerful mechanism for defining a new class by stating only the differences (in terms of features) between the new class and another class that we've already established
  - Assume that we've developed a Student class that is used in a number of applications: SRS, Student Billing System, Alumni Relations System
  - A new requirement has just arisen for modeling graduate students as a special type of student with:
    - Undergraduate degree previously received
    - What institution he/she received the degree from

# Inheritance, cont.

- To appreciate the power of inheritance, let's first look at how we might have to approach the new requirement for tracking Graduate Student information without inheritance



# Without Inheritance

- Approach #1: add the new features into the Student class, along with some way to differentiate whether a student is an undergraduate or a graduate (a boolean "flag")

```
public class Student {  
    private String name;  
    private String idNumber;  
    private String address;  
    // etc. for original attributes  
    private String undergraduateDegree;  
    private String undergraduateInstitution;  
    // Extra attribute (boolean flag).  
    private boolean isGraduate;  
}
```

# Without Inheritance, cont.

- Approach #1, cont.: since the new attributes are only relevant for graduate students, their values will be undefined (null) for undergraduates; we'll have to add extra logic to guard for this

```
public void display() {  
    System.out.println(name);  
    System.out.println(idNumber);  
    // etc. for original attributes  
    if (isGraduate)  
        System.out.println(undergraduateDegree);  
    if (isGraduate)  
        System.out.println(undergraduateInstitution);  
}
```

## Without Inheritance, cont.

- With approach #1, we're modifying code that was already tested and deployed in numerous other applications -- will have to retest carefully
- What if yet another requirement arises, this time to track place of employment for yet another type of student called a Co-Op?

# Without Inheritance, cont.

- We'd have to add even more attributes and another boolean "flag" to the Student class, making it even more complex:

```
public class Student {  
    private String name;  
    private String idNumber;  
    private String address;  
    // etc. for original attributes  
    private String undergraduateDegree;  
    private String undergraduateInstitution;  
    private boolean isGraduate;  
    private String placeOfEmployment;  
    private boolean isCoOp;  
}
```

# Without Inheritance, cont.

```
public void display() {  
    System.out.println(name);  
    System.out.println(idNumber);  
    // etc. for original attributes  
    if (isGraduate)  
        System.out.println(undergraduateDegree);  
    if (isGraduate)  
        System.out.println(undergraduateInstitution);  
    if (isCoOp)  
        System.out.println(placeOfEmployment);  
}
```

- Over time, as our requirements evolve, this code would more and more convoluted -> "spaghetti code"

## Without Inheritance, cont.

- One of the biggest philosophical problems with approach #1 is that we are no longer modeling a single abstraction with our Student class
  - An abstraction is supposed to have just the right number of features, no less, no more, to model a particular real world concept
  - We're trying to model multiple different concepts -- undergraduate student, graduate student, co-op student, etc. -- with a single abstraction

# Without Inheritance, cont.

- Approach #2: "clone" the Student class to create a GraduateStudent class, and add the new features only to the latter

```
public class GraduateStudent {  
    // We've copied the Student attributes ...  
    private String name;  
    private String idNumber;  
    private String address;  
    // ... and then added two more.  
    private String undergraduateDegree;  
    private String undergraduateInstitution;  
  
    // The same would be true for all of the  
    // get/set methods.
```

## Without Inheritance, cont.

- The problem with approach #2 is that we've duplicated an awful lot of code
  - At a minimum, the get/set methods for all of the common attributes along with the attribute data structure
- If we later decide to change how we handle students' idNumbers, as an example -- perhaps switching from String to int as a data type -- we now have to fix the code in two different places
- What happens when we add a third type of student (Co-op, as before)? A fourth?



# Inheritance, cont.

- Using the Java reserved word 'extends', we can define a Graduate Student as a special type of Student, having two extra attributes:

```
public class GraduateStudent extends Student {  
    // Declare two new attributes above and beyond  
    // what the Student class declares ...  
    private String undergraduateDegree;  
    private String undergraduateInstitution;  
  
    // ... and their accessor/modifier methods.  
    public String getUndergraduateDegree() {  
        return undergraduateDegree;  
    }  
  
    public void setUndergraduateDegree(String u) {  
        undergraduateDegree = u;  
    }  
}
```

# Inheritance, cont.

```
public String getUndergraduateInstitution() {  
    return undergraduateInstitution;  
}  
  
public void setUndergraduateInstitution(String u) {  
    undergraduateInstitution = u;  
}  
}
```

- We automatically retain all of the features of Student on top of these explicit features
- It is as if we've 'plagiarized' the code for the attributes and methods of the Student class, and inserted it into GraduateStudent, but without the fuss of actually having done so

# Inheritance, cont.

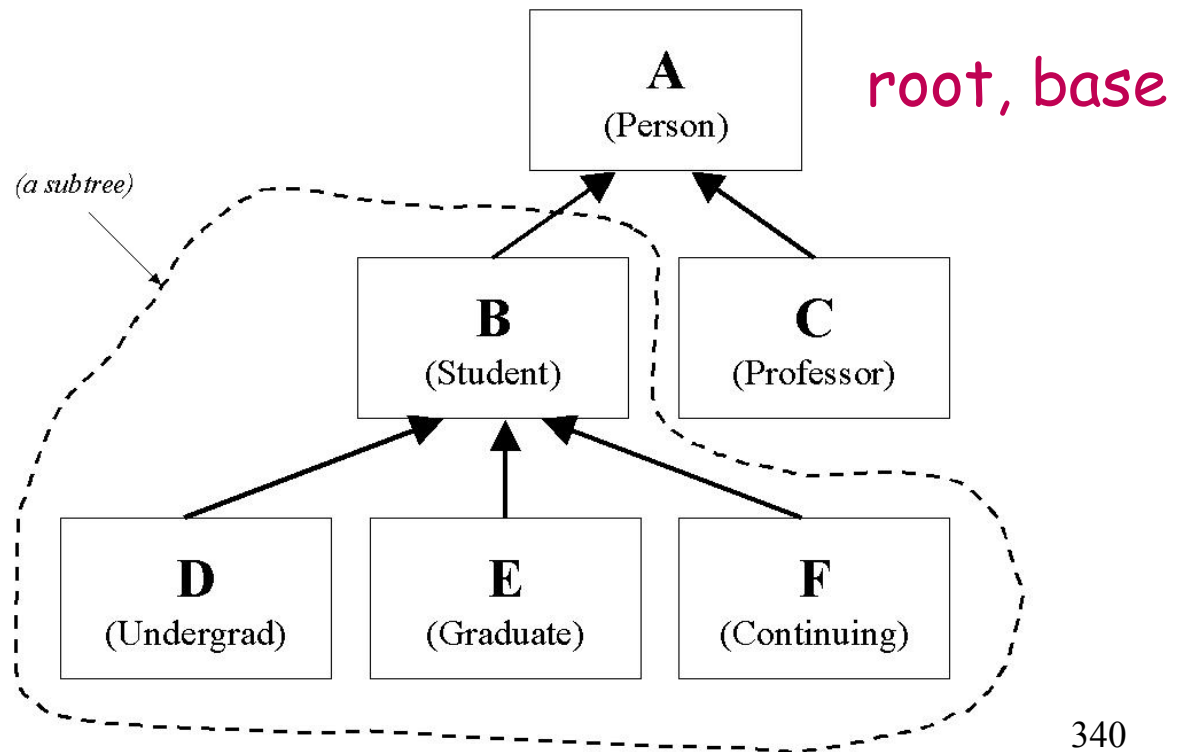
**This code is implied by inheritance**

```
public class GraduateStudent extends Student {  
    { private String name;  
      private String idNumber;  
      private String undergraduateDegree;  
      private String undergraduateInstitution;  
      public String getName() {  
          return name;  
      }  
      public void setName(String n) {  
          name = n;  
      }  
      // etc.  
      public String getUndergraduateDegree() {  
          return undergraduateDegree;  
      }  
      // etc.  
}
```

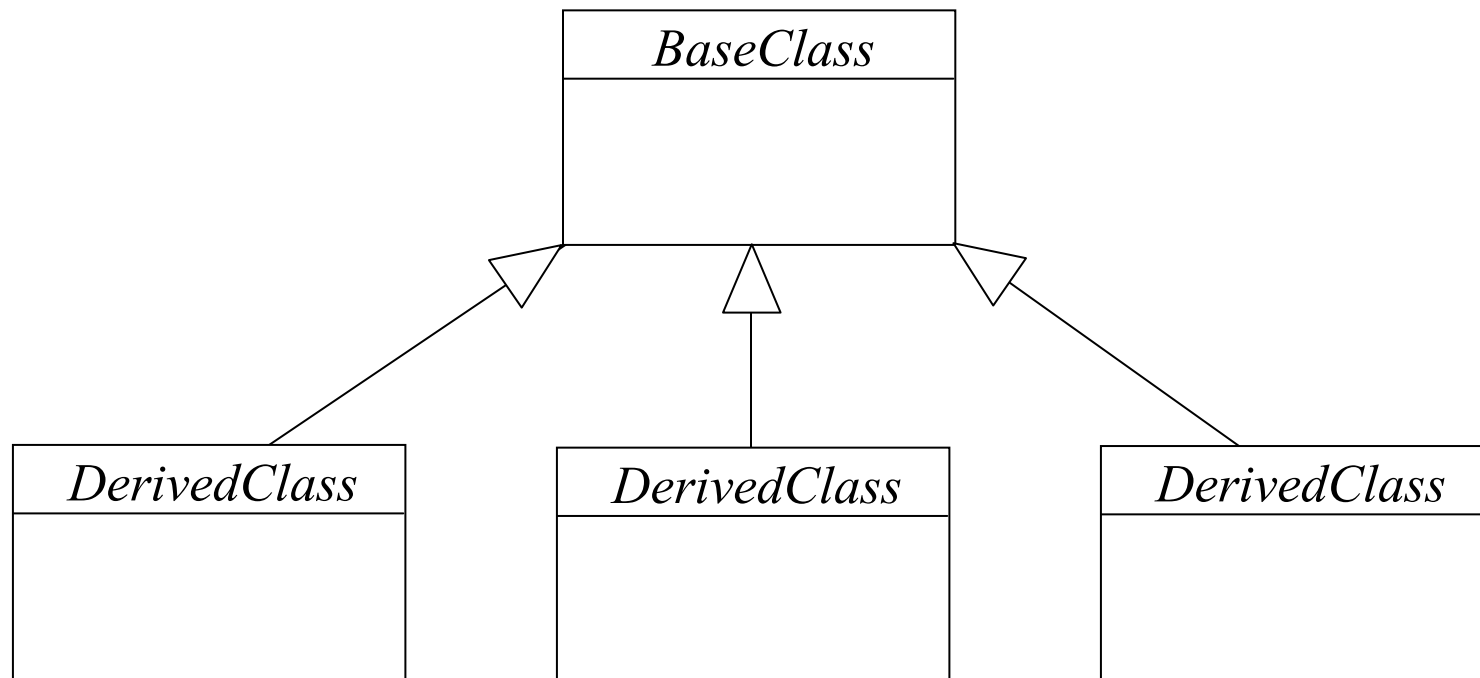
# Class Hierarchy

- Over time, we build up an inverted 'tree' of classes that are interrelated through inheritance

parent class,  
superclass,  
ancestor;  
child class,  
subclass,  
descendant

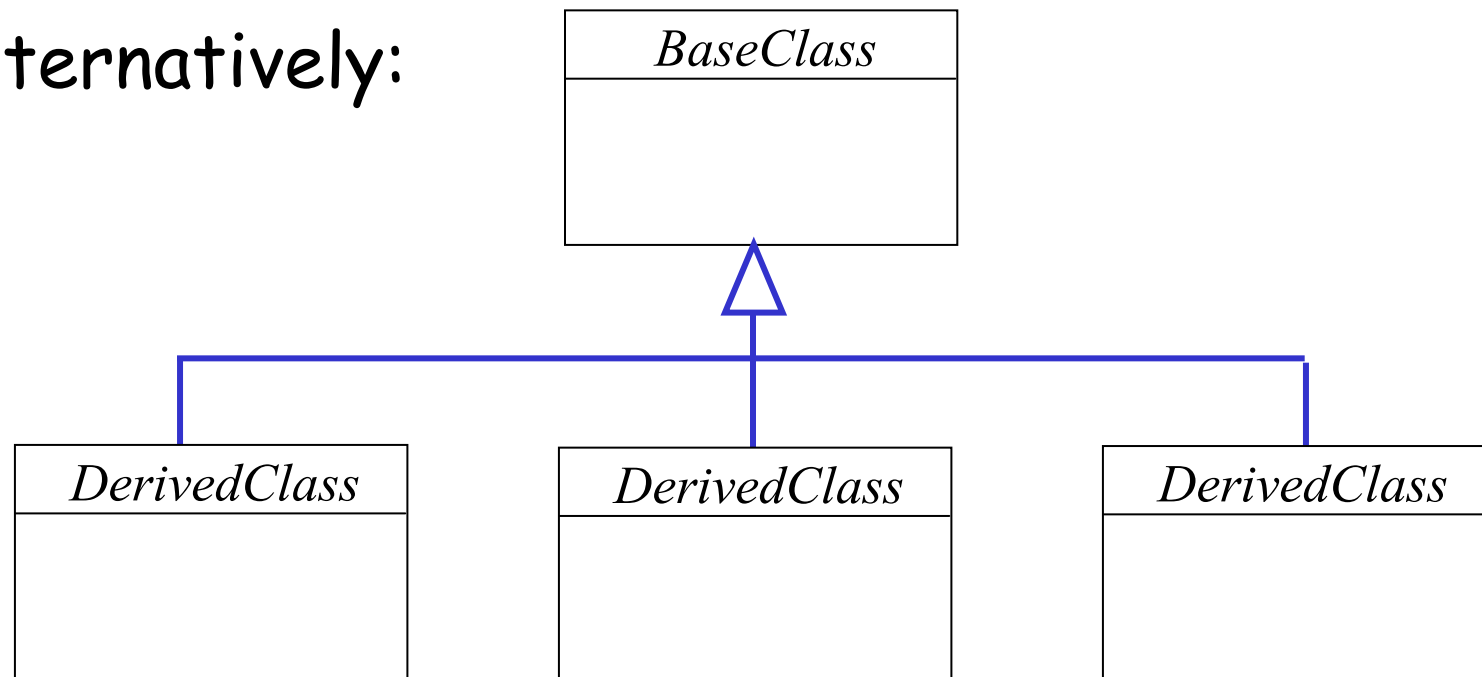


# Representing Inheritance in UML Notation



# Representing Inheritance in UML Notation, cont.

Alternatively:



# The "Is A" Relationship

- Inheritance is often referred to as the "is a" relationship, because any instance of a subclass of class A is, by definition, simultaneously an instance of A, as well
  - If Student is a subclass of Person, then a Student IS A Person
  - Anything that we say about a Person's behavior or data structure **MUST** also be true of a Student for Student to be a proper subclass of Person

# The Object Class

- In some OO languages (Java, Smalltalk), there is a built-in class called `Object` which is the implied root of all class hierarchies, user defined or otherwise
- This has many implications, which we'll discuss along the way
  - Imparts common set of characteristics to all objects
  - Allows all objects to be interchangeable, regardless of type, in certain contexts



# Visibility Review

- We learned that declaring a feature to have **public** access means that the feature so marked is accessible by client code throughout the application

```
public class Person {  
    public String name;  
}
```

---

```
Person p = new Person();  
p.name = "Fred";
```

- When a public feature is inherited by a subclass, it remains public

# Visibility Review, cont.

- We also learned that **private** access means that a feature is accessible from within a class's own methods, but not from client code

```
public class Person {  
    public String name;  
    private String ssn;  
  
    public void setSsn(String s) {  
        ssn = s;    // OK  
    }  
}
```

---

```
Person p = new Person();  
p.ssn = "123-45-6789";    // NOT OK
```

# Visibility Review, cont.

- As it turns out, a private feature is *so* private that it is not directly accessible to a subclass that inherits it, either
  - Like an internal organ

```
public class Person {  
    public String name;  
    private String ssn;
```

---

```
public class Student extends Person {  
    // All attributes of Person are inherited.  
  
    public void display() {  
        System.out.println(name);  
        System.out.println(ssn); // NOT OK!
```

# Visibility Review, cont.

- While a subclass indeed inherits all private features of its parent class -- they become part of the structure of the subclass -- they are effectively "invisible" to the subclass
- How do we access such inherited yet invisible features? Via the public features that we've inherited!

```
public class Student extends Person {  
    public void display() {  
        System.out.println(name);  
        System.out.println(getSsn()); // OK!
```

# Visibility Expanded

- If we want a feature to be private to client code but directly accessible by subclasses, we declare it to be **protected** (a third visibility category)

```
public class Person {  
    public String name;  
    private String ssn;  
    protected String address;  
}
```

---

```
public class Student extends Person {  
    public void display() {  
        System.out.println(name);  
        System.out.println(getSsn());  
        System.out.println(address); // OK!  
    }  
}
```

---

```
// Client code:  
Student s = new Student();  
s.address = "123 Main Street"; // NOT OK!
```

# Generalization/Specialization

- Specialization: growing a class hierarchy downward
- Generalization: growing a class hierarchy upward
  - Recognize similar features in two or more classes, and consolidate these in a common superclass
  - E.g., create a Person class after the fact to house common features of Student and Professor

# Two Ways to Specialize

- #1: We may **extend** the base class by **adding features**
  - We've already seen this in `GraduateStudent`, where we added two attributes and four methods beyond what we inherited from `Student`

# Two Ways to Specialize, cont.

- #2: We may **specialize** the way that a subclass performs one or more inherited **services**
  - For example, when a 'generic' student enrolls for a course, the student may first need to ensure that:
    - He or she has taken the necessary prerequisite courses
    - The course is required for the degree the student is seeking
  - When a *graduate* student enrolls for a course, he/she may need to do both of these things as well as to:
    - Ensure that his/her graduate committee approves the course



# Overriding

- Specializing the way that a subclass responds to a given message as compared with the way that its parent class would respond to the same message
  - We do so by explicitly programming a method in the subclass with an identical method signature, 'rewiring' how a method works internally
  - This version masks/replaces the inherited version

# Overriding, cont.

```
public class Student {  
    // Attributes.  
    protected String name;  
    protected String studentId;  
    protected String majorField;  
    protected double gpa;  
  
    public void print() {  
        // We print out all the attributes that the  
        // Student class knows about.  
        System.out.println("Student Name:  " + name  
            + "\n" + "Student No.:  " + studentId  
            + "\n" + "Major Field:  " + majorField  
            + "\n" + "GPA:  " + gpa);  
    }  
}
```

# Overriding, cont.

```
public class GraduateStudent extends Student {  
    // We add two new attributes.  
    private String undergraduateDegree;  
    private String undergraduateInstitution;  
    // (Plus the get/set methods (not shown).)  
  
    // No methods overridden ... what will happen?  
}
```

# Overriding, cont.

```
public class GraduateStudent extends Student {
    // Attributes.
    private String undergraduateDegree;
    private String undergraduateInstitution;

    // The Student version will be inherited "as is" ...
    public void print() {
        // We print out all the attributes that the
        // Student class knows about.
        System.out.println("Student Name:  " + name
            + "\n" + "Student No.:  " + studentId
            + "\n" + "Major Field:  " + majorField
            + "\n" + "GPA:  " + gpa);
    }
    // ... and knows nothing about the new attributes!
}
```

# Overriding, cont.

```
public class GraduateStudent extends Student {  
    // Attributes.  
    private String undergraduateDegree;  
    private String undergraduateInstitution;  
  
    // We must explicitly override the method if we want  
    // to alter the behavior.  
    public void print() {  
        System.out.println("Student Name:  " + name + "\n" +  
            "Student No.:  " + studentId + "\n" +  
            "Major Field:  " + majorField + "\n" +  
            "GPA:  " + gpa + "\n" +  
            "Undergrad. Deg.:  " + undergraduateDegree +  
            "\n" + "Undergrad. Inst.:  " +  
            undergraduateInstitution);  
    }  
}
```

# Reusing Base Class Behaviors With 'super'

- Having to duplicate code between generations of classes is inefficient
- The reserved word 'super' is used when we want to refer to the parent class version of some object from within one of its methods
  - Like talking to an object's "alter ego"!

# Overriding, cont.

```
public class GraduateStudent extends Student {
    // Attributes.
    private String undergraduateDegree;
    private String undergraduateInstitution;

    // We must explicitly override the method.
    public void print() {
        // Reuse parent's version "as is".
        super.print();

        // Additional logic.
        System.out.println("Undergrad. Deg.:  " +
            undergraduateDegree +
            "\n" + "Undergrad. Inst.:  " +
            undergraduateInstitution);
    }
}
```

# Reusing Base Class Behaviors With 'super', cont.

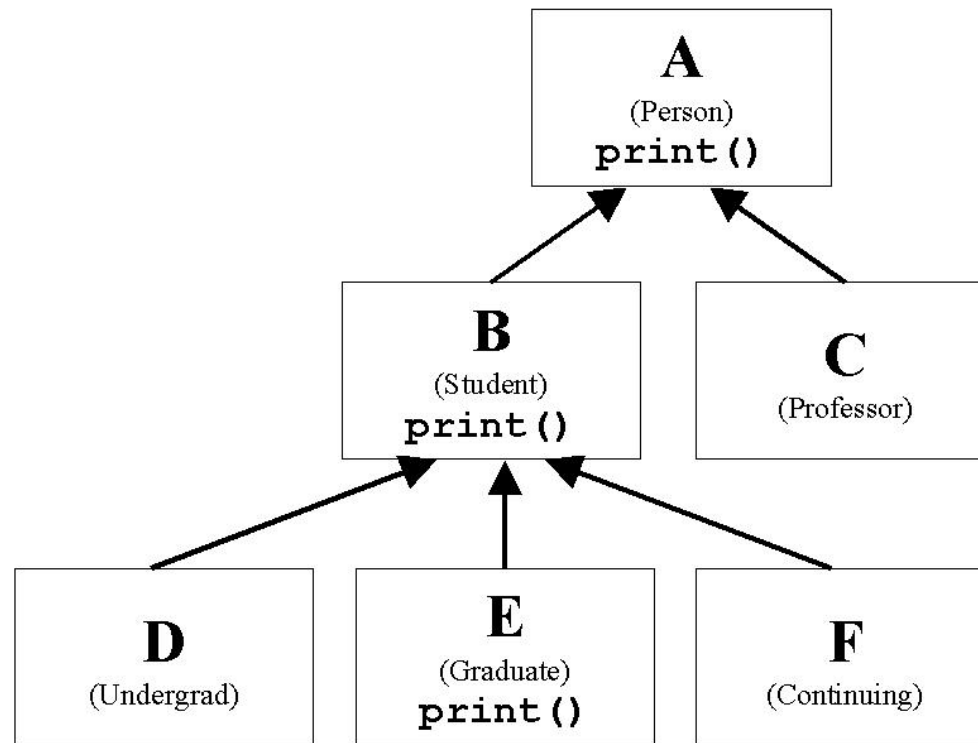
- In the case of a method that takes arguments, we must pass these arguments through when we invoke the "super" version:

```
public void foobar(String x, int y) {  
    super.foobar(x, y);  
    // etc.  
}
```

- We are sending a message to our alter ego



# Overriding, cont.



# Inheritance Simplifies Code Devel. and Maint.

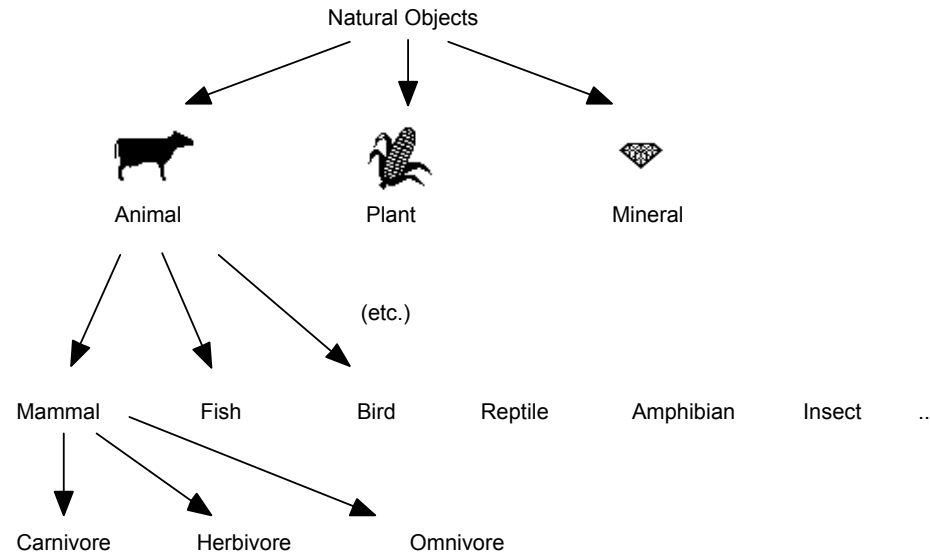
- Inheritance is perhaps one of the most powerful and unique aspects of an OOPL
  - Through inheritance, we can reuse and extend code that has already been thoroughly tested without modifying (and risking breaking) it
    - Even classes for which we don't own the source code! (e.g., Sun's built in Java classes)
  - This is one of the significant ways to achieve productivity with an object-oriented language

# Inheritance Simplifies Code Devel. and Maint.

- The total body of code for a given application is significantly reduced as compared with the traditional non-OO approach, and thus easier to maintain
  - Think of how concise the *GraduateStudent* class was -- it only contained the essence of what make it different from generic *Student*

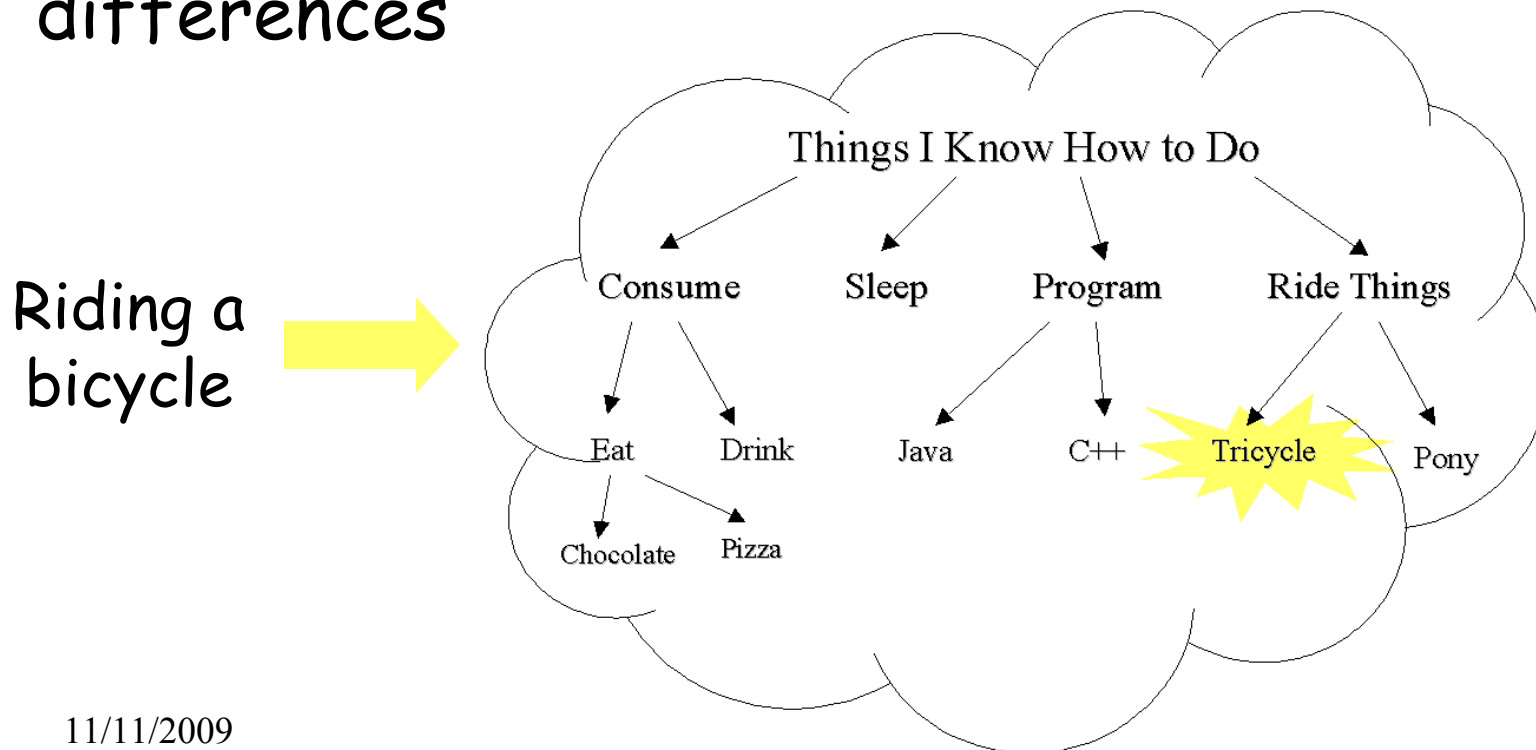
# Classification, Revisited

- Organizing classes in an inheritance hierarchy is intuitive
- As we saw earlier, we naturally organize concepts into hierarchies ...



# Searching the Hierarchy

- When we are learning a new concept, we search our mental hierarchy for as similar a concept as we can find, so as to focus only on the differences



# Overloading

- Overloading is a language mechanism supported by non-OO languages like C as well as by OOPL
- Overloading allows two or more different methods\* belonging to the same class to have the *same* name as long as they have *different argument signatures*
  - Neither the return type of the method nor the argument names enter into the picture with overloading

# Overloading

- Student class example - five different arg. signatures

```
boolean print();  
void print(String fileName);  
void print(int detailLevel);  
void print(int detailLevel, String fileName);  
int print(String reportTitle, int maxPages);
```

- Choose among these five 'flavors' of print() based on what form of message we send to a Student object:

```
Student s = new Student();  
s.print("output.rpt");  
s.print(2);  
s.print(2, "output.rpt");  
// etc.
```

- We see that neither the *names* of the args nor the *return type* of the method are evident in a message

# Specialization "Don'ts (Can'ts)"

- You should not change the 'semantics' - i.e. the intention, or meaning - of a feature
- You cannot 'override' (nor overload) an attribute
- You cannot physically eliminate ("uninherit") features, nor should you effectively eliminate them by ignoring them
  - To attempt to do breaks the spirit of the 'is a' hierarchy
  - Inheritance mandates that all features of all *ancestors* of a class 'X' must also apply to class X itself in order for X to truly be a proper subclass



# Specialization "Don'ts", cont.

- You cannot change the signature of a method -- this results in overloading

```
public class Student {  
    public void display() { ... }  
}
```

---

```
public class GraduateStudent extends Student {  
    public void display(String fileName) { ... }  
  
    // We STILL inherit the parent's version - we now have  
    // TWO display methods!  
    public void display() { ... }  
}
```

# Exercise #6

# Constructors

- When we talked about instantiating objects, you may have been curious about the interesting syntax involved with the 'new' operator:

```
Student x = new Student();
```

- Why are there parentheses tacked onto the end of the statement? We are actually invoking a special type of method called a **constructor**
- A constructor literally constructs (instantiates) a brand new object by allocating enough program memory to house the object's attributes

# Constructors, cont.

- The syntax of a constructor signature is a bit different from other methods:

`public`                          Student   (String id, String name)

The diagram illustrates the syntax of a constructor signature. It shows the code `public` followed by a blank line, then `Student` (underlined), and then `(String id, String name)` (underlined). Three arrows point from explanatory text below to these parts: a red arrow from 'NO return type! (implied)' to the blank line; a black arrow from 'method name MATCHES the class name' to `Student`; and a blue arrow from 'optional argument list (may be empty ( ))' to the parentheses and their contents.

NO  
return type!  
(implied)

method name  
MATCHES the  
class name

optional argument list  
(may be empty ( ))

# Constructors, cont.

- The argument list of a constructor may be used to specify which attributes you want to initialize, if any, at the time that an object is being instantiated

// Instead of:

```
Student s = new Student();  
s.setName("Fred Schnurd");  
s.setSsn("123-45-6789");  
s.setMajor("MATH");  
// etc.
```

// We have:

```
Student s = new Student("Fred Schnurd",  
                        "123-45-6789", "MATH");
```

# Constructors, cont.

- The body of the constructor would then look as follows:

```
class Student {  
    // Attributes.  
    String name;  
    String ssn;  
    String major;  
    // etc.  
  
    public Student(String n, String s, String m){  
        name = n; // or:  name = new String(n);  
        ssn = s;  
        major = m;  
    }  
  
    // etc.
```

# Constructors, cont.

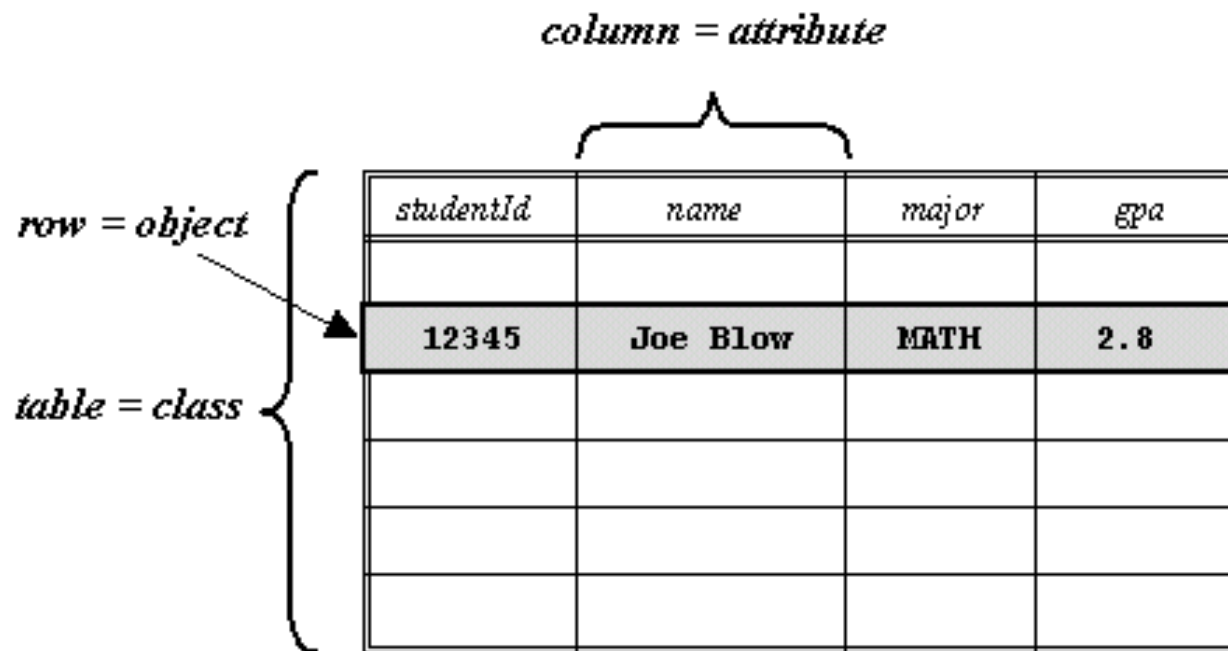
- We may actually write many different constructors for the same class which take differing numbers of arguments:

```
public class Student {  
    // Attributes.  
    String name;  
    String ssn;  
    String major;  
    // etc.  
    public Student(String n, String s, String m) {...}  
    public Student(String n, String s) {...}  
    public Student(String s) {...}
```

- Another example of **overloading**

# Objects and Databases

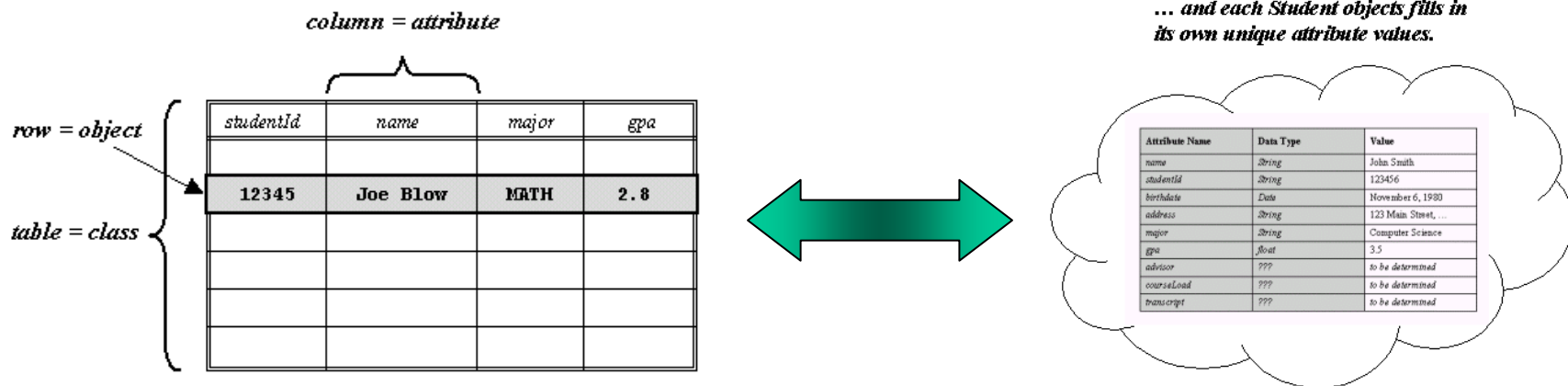
- There is a tendency to relate the concept of an object to that of a record in a database





# Objects and Databases, cont.

- We often persist/restore the state of an object with a database



# Constructors and Databases

- Instead of passing arguments into a constructor to initialize the attributes for an object, we may have the constructor fetch data from a database

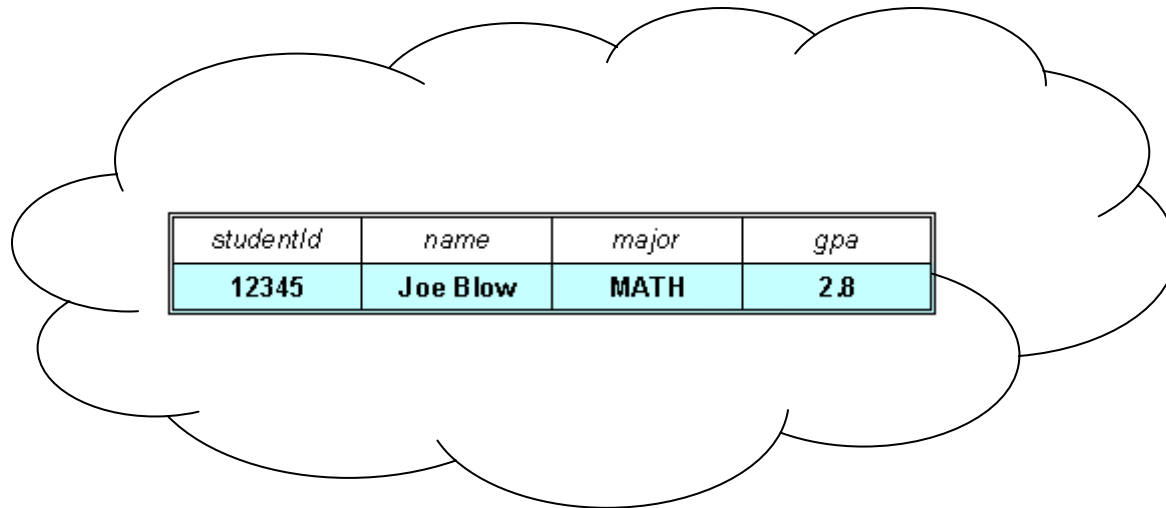
```
public class Student {  
    // Attribute details omitted.  
    // Constructor.  
    public Student(String ssn) {  
        use ssn as a primary key to look up this student's  
        record in the database's Student table;  
        if (record found) {  
            retrieve record and initialize all attributes  
            based on this record;  
        }  
    }  
}
```

# Constructors and Databases, cont.

- As we'll discuss later, however, we typically don't want to clutter up our model classes with database access logic
- Rather, we'll want to isolate such logic in one or more **data access layer classes**
- This refers to the design principle of **model - data layer separation**

# Objects and Databases, cont.

- Each in-memory object houses one record's worth of data



# Constructors, cont.

- One other 'oddity' with respect to constructors, as compared with other methods, is that invoking them does not involve dot notation:

```
Professor p = new Professor();
```

- This is because we are not requesting a service of a particular object, but rather are requesting that a new object be crafted from 'thin air' by the programming environment

can be THOUGHT OF as: `Professor p = Professor.Professor();`

# Default Constructor

- If we don't explicitly declare any constructor for a class, Java provides a default "no args" constructor for us:

```
Student s = new Student();
```

- If we need to do something special when an object is created, we may override this method signature explicitly:

```
public class Student {  
    // Explicitly override the default constructor.  
    public Student() {  
        do whatever special processing that we deem  
        necessary to construct a Student  
    }  
}
```

# Default Constructor, cont.

- If we invent any constructors of our own, we lose the default "no args" constructor

```
public class Person {  
    protected String name;  
  
    // Constructor.  
    public Person(String n) {  
        name = n;  
    }  
  
    // Default constructor is LOST!  
}
```

- In this case, we can no longer do the following in client code:

```
Person p = new Person();
```

# Default Constructor, cont.

- If we want a "no args" constructor, we must explicitly add one back:

```
public class Person {  
    protected String name;  
  
    public Person(String n) {  
        name = n;  
    }  
  
    public Person() {  
        name = "?";  
    }  
}
```

- This is generally a good practice, for reasons that we'll see in a moment



# Reusing Constructor Code

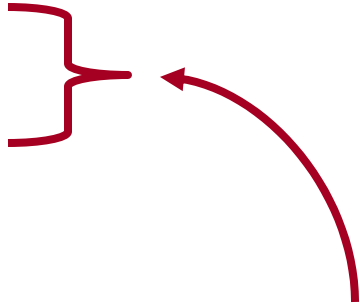
- We can reuse the code of one constructor from within another constructor in the same class via `this(...)` to avoid code duplication

```
public class Student {  
    // Attributes omitted - see earlier example.  
  
    // Constructor #1.  
    public Student() {  
        name = "?";  
        ssn = "?";  
        major = "TBD";  
    }  
}
```

# Reusing Constructor Code, cont.

- We can reuse the code of one constructor from within another constructor in the same class via `this(...)` to avoid code duplication

```
public class Student {  
    // Attributes omitted - see earlier example.  
  
    // Constructor #1.  
    public Student() {  
        name = "?";  
        ssn = "?";  
        major = "TBD";  
    }  
  
    // Constructor #2.  
    public Student(String s) {  
        this(); // perform code of Student()  
        ssn = s; // whatever extra logic makes sense  
    }  
}
```

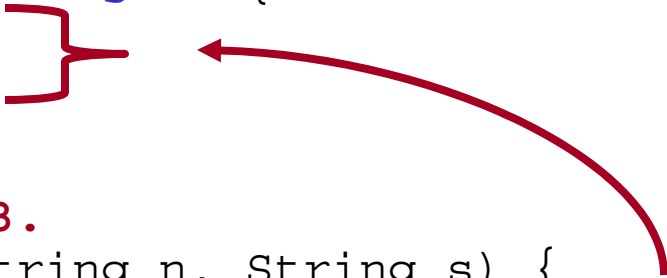


# Reusing Constructor Code, cont.

```
// (Repeated from previous slide.)
// Constructor #2.
public Student(String s) {
    this();
    ssn = s;
}

// Constructor #3.
public Student(String n, String s) {
    this(s); // perform code of Student(String s)
    name = n;
    perhaps some extra steps ...
}

// etc.
```

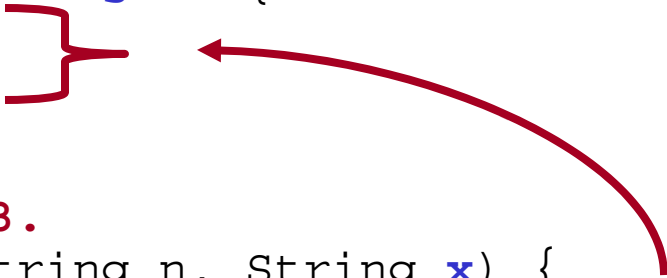


# Reusing Constructor Code, cont.

```
// (Repeated from previous slide.)
// Constructor #2.
public Student(String s) {
    this();
    ssn = s;
}

// Constructor #3.
public Student(String n, String x) {
    this(x); // perform code of Student(String s)
    name = n;
    perhaps some extra steps ...
}

// etc.
```



# Constructors and Inheritance

- There are number of complicating factors that we need to be aware of with respect to constructors whenever we are working with an inheritance hierarchy

# Constructors and Inheritance

- Factor #1: Constructors are not inherited as other methods are
  - If we write a constructor for Person that takes a single String argument:

```
public class Person {  
    protected String name;  
  
    // Constructor.  
    public Person(String n) {  
        name = n;  
    }  
}
```

---

```
// Client code:  
Person p = new Person("Joe");
```

# Constructors and Inheritance

- It is not true that the same constructor signature will automatically be available for a *subclass* of *Person*

```
public class Student extends Person { ... }
```

---

```
// Client code:
```

```
Person p = new Person("Joe");
```

```
Student s = new Student("Fred"); // NOT AUTOMATIC!
```

# Constructors and Inheritance

- We'd have to explicitly program a constructor with that same signature in the subclass if we wanted such a constructor signature available to the subclass

```
public class Person {  
    protected String name;  
  
    public Person(String n) {  
        name = n;  
    }  
}
```

---

```
public class Student extends Person {  
    public Student(String n) {  
        name = n;  
    }  
}
```



# Constructors and Inheritance

- If we want to reuse a parent's constructor within a subclass constructor, we employ the same "super" keyword that we learned about earlier, but with a different syntax:

```
public class Person {  
    protected String name;  
  
    public Person(String n) {  
        name = n;  
    }  
}
```

(match argument signatures)

---

```
public class Student extends Person {  
    public Student(String x) {  
        super(x); // reusing parent constructor  
        // with same arg. signature  
    }  
}
```

# Constructors and Inheritance

## - Another example:

```
public class Person {  
    protected String name;  
    protected String address;  
    public Person(String n, String a) {  
        address = a;  
        name = n;  
    }  
    public Person(String n) {  
        name = n;  
    }  
}
```

(match argument signatures)

---

```
public class Student extends Person {  
    String studentId;  
    public Student(String x, String i) {  
        super(x);  
        studentId = i;  
    }  
}
```

# Constructors and Inheritance

- One final example:

```
public class Person {  
    protected String name;  
    protected String address;  
    public Person(String n, String a) {  
        address = a;  
        name = n;  
    }  
    public Person(String n) {  
        name = n;  
    }  
}
```

(match argument signatures)

---

```
public class Student extends Person {  
    String studentId;  
    public Student(String x) {  
        super(x, "?");  
        studentId = "?";  
    }  
}
```

# Constructors and Inheritance

- Factor #2: whenever a constructor for a subclass is called, constructor(s) for all of its ancestor class(es) are also called -- either explicitly, or by default -- in top-to-bottom order
  - This is in keeping with the "is a" nature of inheritance
  - For example, if Student is a subclass of Person, then whenever we call the constructor for a Student, the constructor for a Person also comes into play, because a Student is both

# Constructors and Inheritance

- Case #1: We've written no explicit constructors for either the Person or the Student class, and call the default Student constructor

```
Student s = new Student();
```

- In this case, the default Student constructor automatically calls the default Person constructor as its first step

# Constructors and Inheritance

- Case #2: we've written an explicit constructor for Student

```
public class Student extends Person {  
    String major;  
  
    public Student(String n, String m) {  
        // Although we don't see it in the code,  
        // the first thing that happens in this  
        // constructor is an implied call to the  
        // default Person constructor  
        name = n;  
        major = m;  
    }  
}
```

# Constructors and Inheritance

- Case #2: we've written an explicit constructor for Student

```
public class Student extends Person {  
    String major;  
  
    public Student(String n, String m) {  
        // Although it would be unnecessary to  
        // do so, we could explicitly insert the  
        // following line of code to accomplish  
        // the same step:  
        super();  
        name = n;  
        major = m;  
    }  
}
```

# Constructors and Inheritance

- The default constructor for a superclass will always be implicitly called as the first line of a subclass's constructor ...

```
public Student(String n, String m) {  
    super();    <-- automatically implied  
    name = n;  
    major = m;  
}
```

... unless we explicitly call a different version of the parent's constructor to accomplish code reuse (must be the first line of code)

```
public Student(String n, String u) {  
    super(n);    // assume Person(String n) in Person  
    undergraduateMajor = u;  
}
```



# Constructors and Inheritance

- Either way, a Person constructor will always be called as the first step in constructing a Student
  - There is no getting around this, nor should we want to circumvent this language feature
  - This phenomenon can nonetheless cause us some problems

# Constructors and Inheritance

- Assume we have written the following code:

```
public class Person {  
    protected String name;  
  
    // Constructor.  
    public Person(String n) {  
        name = n;  
    }  
  
    // THE "NO-ARGS" CONSTRUCTOR IS LOST.  
}
```

---

```
public class Student extends Person {  
    String id;  
  
    // NO EXPLICIT CONSTRUCTOR PROVIDED.  
}
```

# Constructors and Inheritance

- We'll have a problem when we try to construct a Student:

```
Student s = new Student();
```

- The Student class's "no args" constructor will automatically try to invoke the "no args" constructor for a Person first, which doesn't exist in this case
- We'll get the following (somewhat cryptic) compilation error:

```
Cannot resolve symbol:  constructor Person()  
location:  class Person  
class Student extends Person  
^
```

# Constructors and Inheritance

- How can we get around this problem?
- When creating an inheritance hierarchy, either:
  - As recommended earlier, ensure that a "no-args" constructor is available for every class in the hierarchy
    - This is the preferred approach, unless it is impossible to initialize a "meaningful" object when no arguments are passed in
  - Provide explicit constructors for all classes in the hierarchy, and use only those -- never rely on the default "no-args" constructor

# Exercise #7

# Review

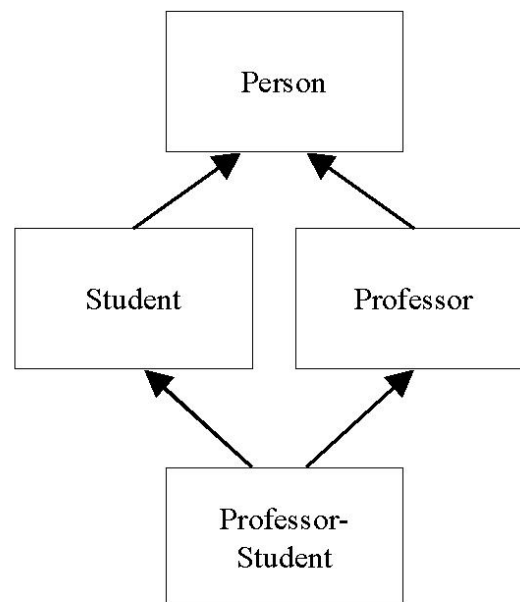
- What Java language mechanisms have we learned about that facilitate code reuse?

# Review

- What Java language mechanisms have we learned about that facilitate code reuse?
  - Reuse of entire classes through inheritance ("**extends**")
  - Reuse of a parent class's method code when overriding via **super.method(...)**
  - Reuse of a parent class's constructor code via **super (...)**
  - Reuse of the code from one constructor from within another constructor in the same class via **this (...)**
  - (Reuse of preexisting classes/interfaces as a whole)
  - (Design pattern reuse)

# Multiple Inheritance

- Multiple inheritance involves more than one parent for a given subclass





# Multiple Inheritance, cont.

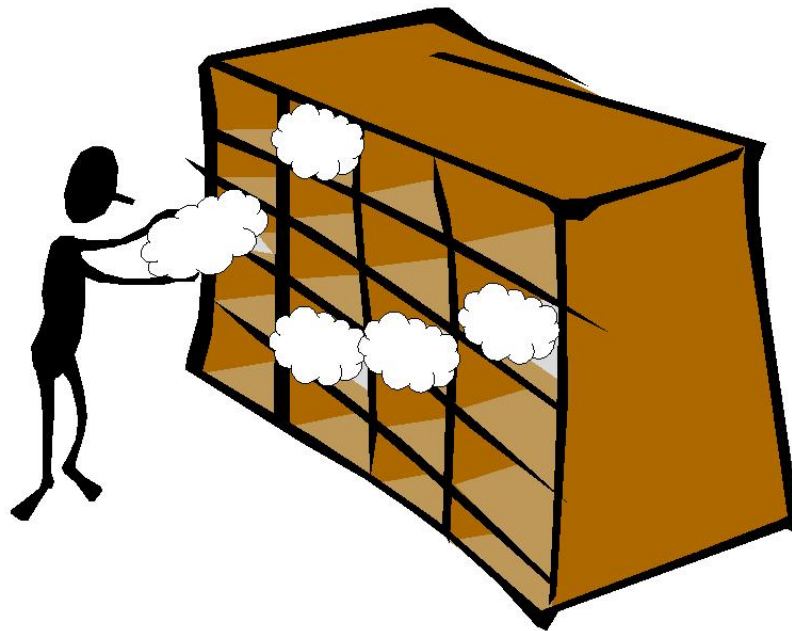
- Deemed problematic
  - Ambiguity with respect to inheriting conflicting features of multiple parents
- Prohibited in Java
- Java interfaces provide a work-around

# Three Features of an OOPL

- ✓ (Programmer creation of) Abstract Data Types (ADTs)
- ✓ Inheritance
- Polymorphism

# Object Collections

## (Chapter 6)

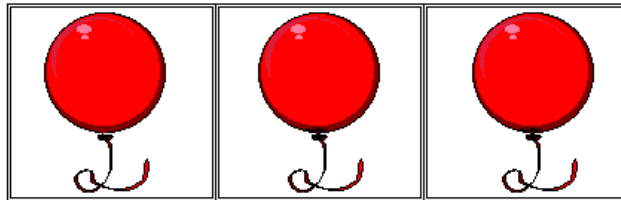


# What are Collections?

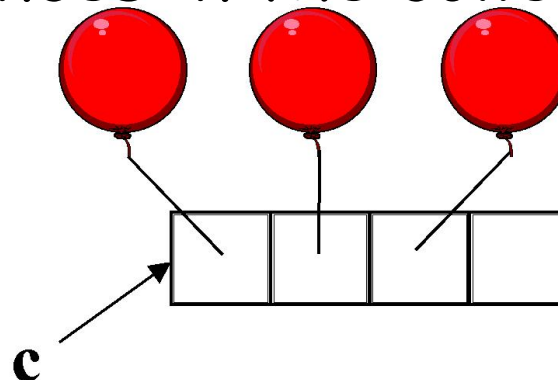
- A collection is a special type of object used to gather up references to other objects as they are created so that we can manage/operate on them collectively and/or individually
  - A Professor object may wish to step through all Student objects registered for a particular Course to compute their grades
  - The Student Registration System may need to step through all of the Course objects to determine which are subject to cancellation
- Think of a collection like an egg carton, and the object (references) it holds like the eggs

# Collections of References

- With some languages (e.g., C++), we actually do physically place the objects themselves into a collection



- With Java, however, we are in actuality storing object references in the collection



# Collections as Objects

- Like any other object, a collection must be instantiated before it is used

```
Vector v; // insufficient  
v = new Vector(); // empty "egg carton" created!
```

- Collections are defined by classes that specify methods for 'getting' and 'setting' their contents

```
// Create a few Student objects and store them in the  
// collection.  
Student a = new Student();  
v.add(a);  
// or:  
v.add(new Student());  
  
// Later in the application:  
// Retrieve a handle on the first Student.  
Student someStudent = (Student) v.elementAt(0);
```

# Collections as Objects, cont.

- Virtually all collections provide methods for:
  - Adding objects\*
  - Removing objects
  - Retrieving specific individual objects
  - Iterating through the objects in some predetermined order
  - Getting a count of the number of objects in the collection
  - Answering true/false questions as to whether a particular object is in the collection or not

\* "objects" throughout is short for "object references"

# Arrays as Simple Collections

- A container of cells for holding like-typed things (ints, chars, references to Students)
- The Java syntax for declaring and using arrays deviates from "typical" object syntax, to make arrays look more like arrays in non-OO languages
- Java arrays are declared in one of two ways:

*datatype[ ] name;*            `int [] x;`            `Student [] y;`

*datatype name[ ];*            `int x[];`            `Student y[];`



# Arrays, cont.

- Instantiated via the 'new' operator:

```
Student[] y = new Student[20];
```

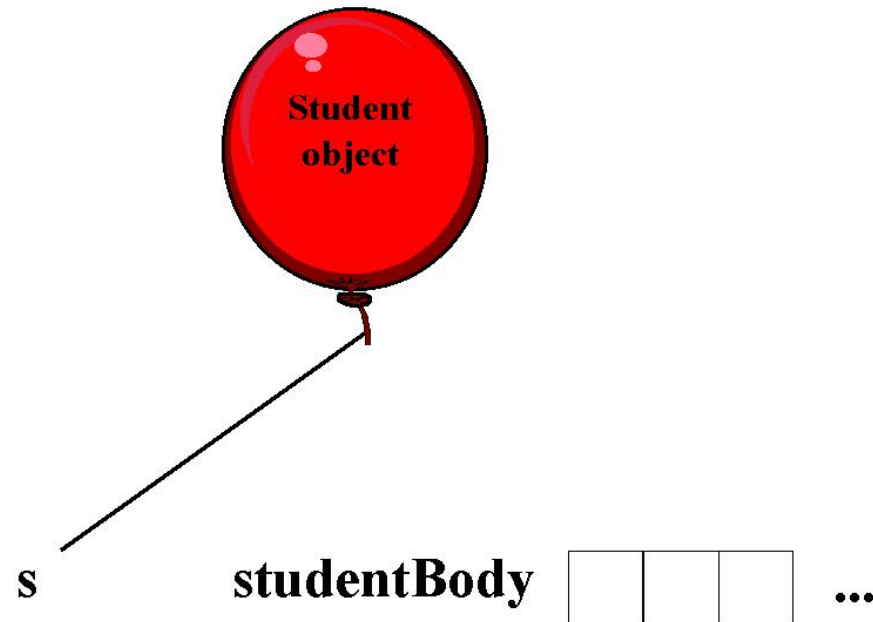
- Although this doesn't look like a typical constructor invocation, we are indeed invoking the *Array* class constructor
- We determine an array's size when we instantiate it, not when it is declared

# Arrays, cont.

- The contents of an array are initialized to their zero-equivalent values when the array is instantiated
  - `int []` filled with zeroes
  - `Student []` filled with `nulls`
- Stuff the array with references

```
Student[] studentBody = new Student[20];
Student s = new Student();
studentBody[0] = s;
// Reuse s!
s = new Student();
studentBody[1] = s;
```

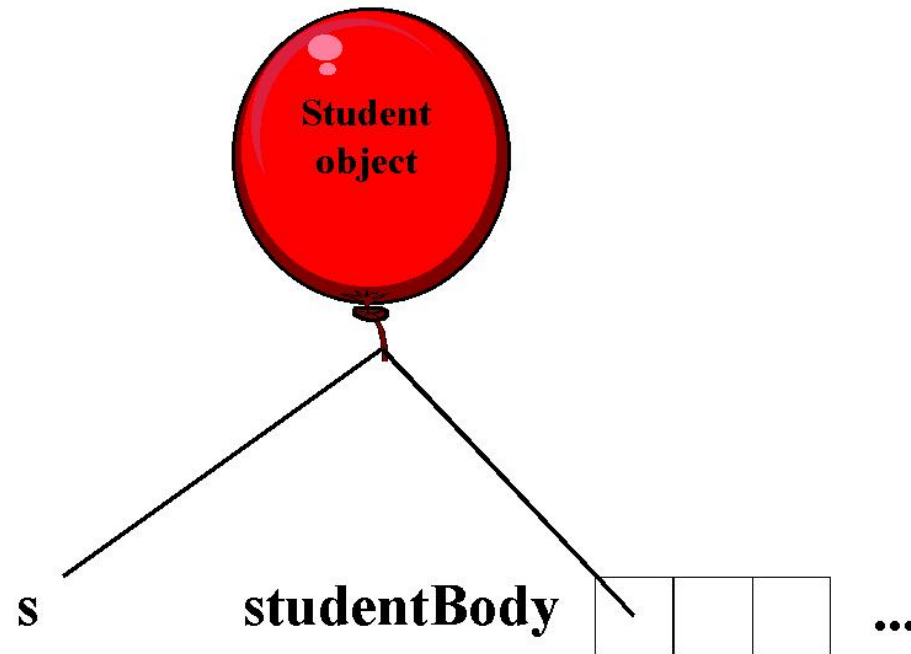
# Arrays, cont.



*A Student object is created and handed to s ...*

```
Student s = new Student();
```

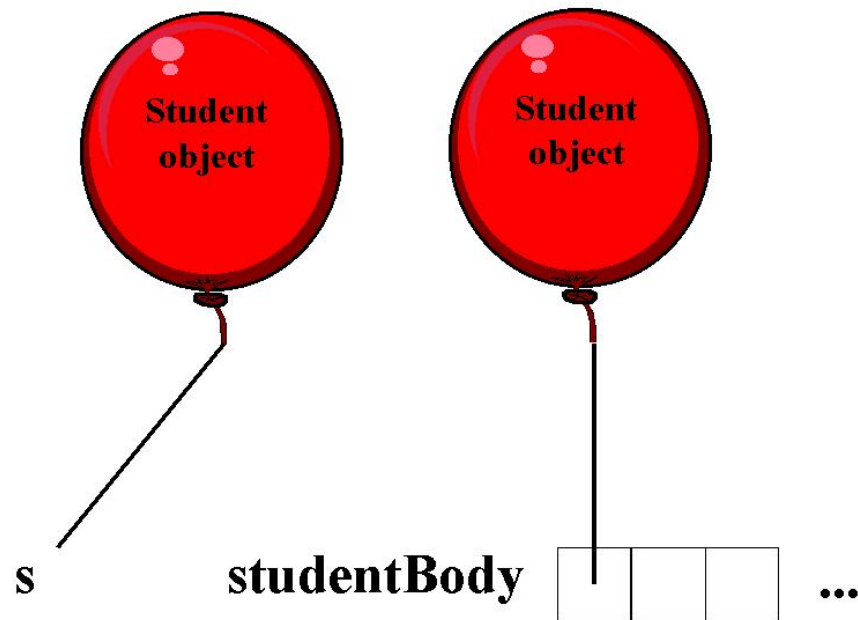
# Arrays, cont.



*... s hands the object's handle off to the array ...*

```
studentBody[0] = s;
```

# Arrays, cont.



*... thus freeing up s to take hold of another new Student!*

```
s = new Student();
```

# Arrays, cont.

- Iterating through an array also involves somewhat odd-looking syntax:

```
for (int i = 0; i < studentBody.length; i++) {  
    // Access the 'ith' element of the array.  
    System.out.println(studentBody[i].getName());  
}
```

- Note length attribute
- length = number of cells, whether empty or not
- Note message passing
- If an array cell is empty (null), a `NullPointerException` arises ("landmine")
  - (we'll see a workaround for this on the next slide)

# Array Limitations

- There are several problems with using an array to hold a collection of objects:
  - Once sized, a 'classic' array cannot typically be expanded
  - It is often hard to predict in advance the number of objects that a collection will need to hold
  - 'Landmine' issues if an array isn't full (we can test for these, however:

```
for (int i = 0; i < studentBody.length; i++) {  
    if (studentBody[i] != null)  
        System.out.println(studentBody[i].getName());  
}
```

# More Sophisticated Collection Types

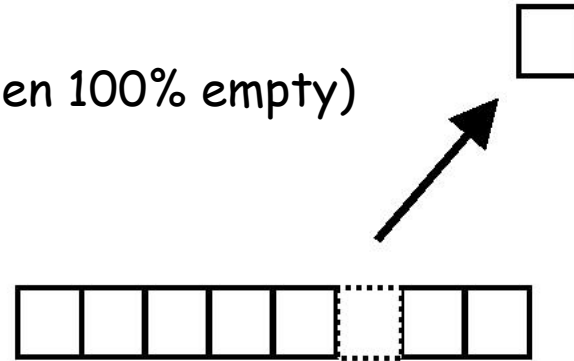
- Ordered List:
  - Similar to an array, in that order is preserved
  - Size does not have to be specified at the time that the collection object is first created
  - Will automatically grow in size as new items are added (true of virtually all collections besides arrays)



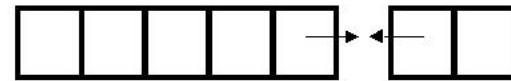
# More Types, cont.

- No "landmines" (except when 100% empty)

*When an item is  
removed from an  
ordered list ...*



*... the 'hole'  
automatically  
closes up!*



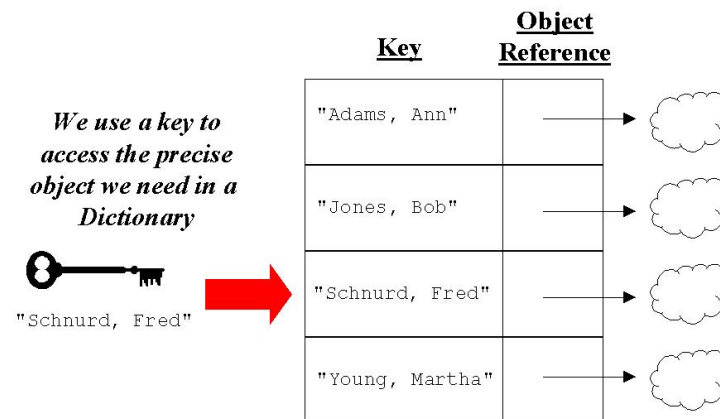
- SRS example: manage a wait list for a course that has become full

# Java-Specific Collections

- Java has numerous built in collection classes that represent ordered lists: Vector, ArrayList, LinkedList, Stack
- We'll use the Vector class in our banking exercise
  - We'll talk about this class in a bit more depth shortly

# More Types, cont.

- Dictionary:
  - Store each object reference along with a unique retrieval key (typically based on attribute values)
  - Retrieve directly by key, or iterate through in key order



## More Types, cont.

- SRS example: a dictionary, indexed on a unique combination of course number plus section number, to manage its semester schedule

# Java-Specific Collections, cont.

- Java has numerous built in collections that implement dictionaries: Hashtable, TreeMap, Hashmap, etc.
- The Hashtable class is used in the SRS code example associated with this book

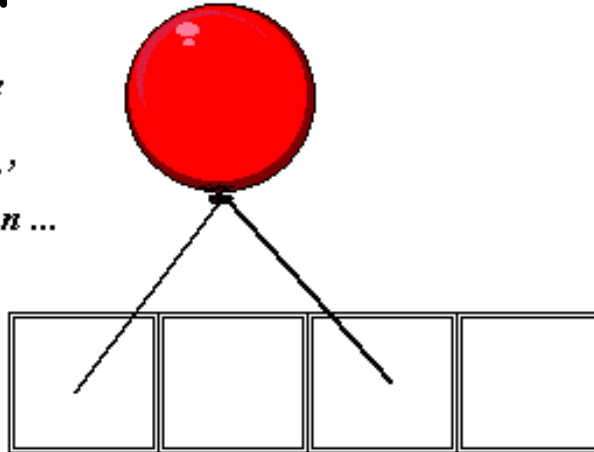
# More Types, cont.

- Set:
  - An unordered collection (like an assortment of differently colored marbles in a sack)
    - We can reach into the sack to pull the marbles out one by one, but the order with which we pull them out is non-deterministic
    - We can step through the entire set to perform some operation on each object, but can't guarantee in what order the objects will be processed
    - We can perform tests to determine whether a given specific object has been previously added to a set or not

## More Types, cont.

- A Set doesn't allow duplicates, whereas most other collection types do:

*The same object may be  
referenced in  
multiple 'compartments'  
within a single collection ...*



*... UNLESS the collection is a set!*

- SRS example: Biology Department majors

# Java-Specific Collections, cont.

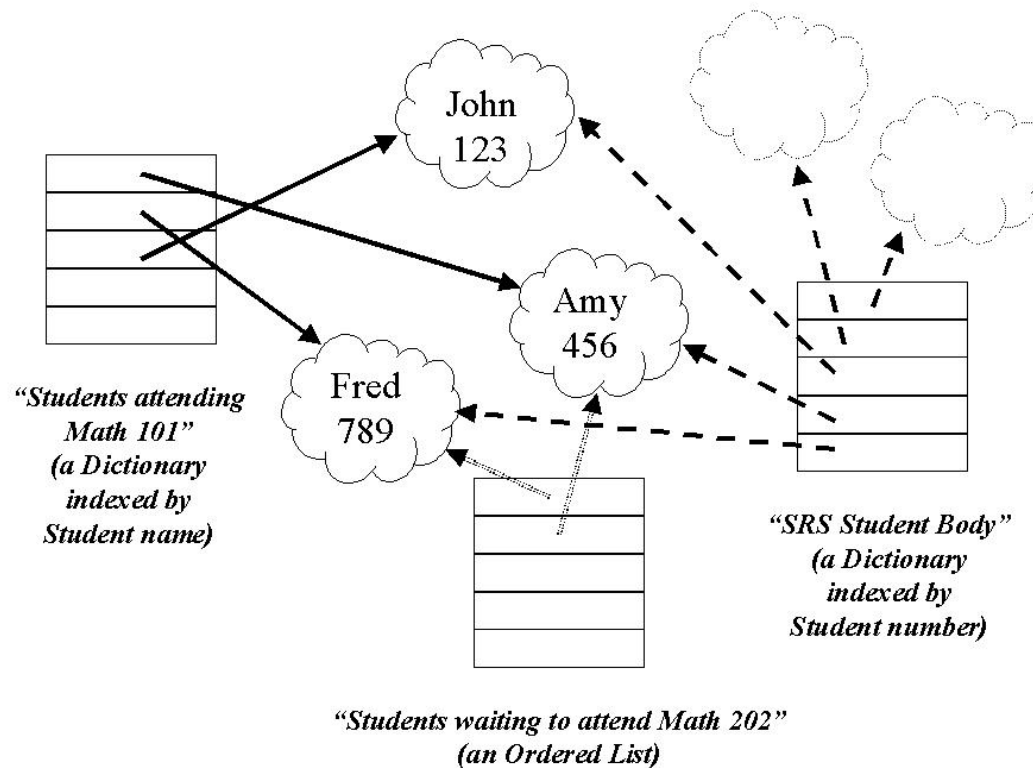
- Java has several built in collections that implement sets: `HashSet`, `TreeSet`, etc.



# Choosing a Collection Type

- Collections are encapsulated, and hence take full advantage of information hiding
  - We don't need to know the private details of how object references are stored internally to a collection
  - We only need to know a collection's public behaviors in order to choose an appropriate collection type for a particular situation and to use it effectively

# Objects May Coexist in Multiple Collections



\* Each object only instantiated once!!! \*

# Enhancements to Collections as of Java 5 (v. 1.5.x)

- Version 1.5.0 of Java (aka Java 5) introduced some significant enhancements with regard to how collections are dealt with
  - Easier to code
  - Enhanced compile time checking at compile time - fewer run time issues
- It's important to also understand the more cumbersome syntax of collections as used with 1.4.x and earlier - we'll cover this first

# Constraining a Collection's Content Type (Java 1.4.x and Earlier)

- Java collection types other than Arrays don't allow you to specify what type of object they will hold when you declare them
  - Virtually all Java collections were designed to hold objects of type Object, the "mother of all classes"

```
// ArrayLists hold generic Objects.  
ArrayList x = new ArrayList(); // No type designated!
```

```
// With arrays, we DO specify a type.  
Student[] s = new Student[100];
```

# Constraining a Collection's Content Type, cont.

- So, we can pretty much put whatever type of object we wish into a collection, because every Java object is descended from Object!
- It is important that a programmer know what the intended (super)type for a collection is going to be, so that only objects of the proper type are inserted in the collection

```
ArrayList x = new ArrayList(); // of Students
```

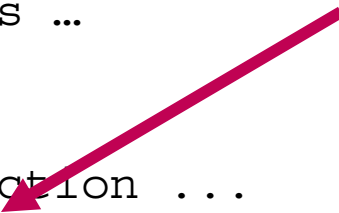
- This will be important when you subsequently attempt to iterate through and process all of the objects in a collection

# Retrieving Objects from a Java Collection

- Objects 'remember' their types when inserted into a collection, but are treated as generic Objects by the collection
- We have to assure the compiler of their type when pulling them back out (by casting):

```
ArrayList x = new ArrayList(); // of Students
Student s1 = new Student();
Student s2 = new Student();
// Insert the Student objects ...
x.add(s1);
x.add(s2);
// Iterate through the collection ...
for (int i = 0; i < x.size(); i++) {
    Student s = (Student) x.get(i); // cast
    // etc.
```

(note: size()  
METHOD vs.  
length ATTRIB.)



# Collections of Supertypes

- If we create a collection intended to hold objects of a given type - e.g., Person - then it makes perfect sense to insert a mixture of objects of type Person or of *any* of the *subtypes* of Person
  - This is by virtue of the 'is a' nature of inheritance

# Collections of Supertypes

```
ArrayList srsUsers = new ArrayList(); // of Person objects

Professor p = new Professor();
UndergraduateStudent s1 = new UndergraduateStudent();
GraduateStudent s2 = new GraduateStudent();

// Add a mixture of professors and students; all have
// Person as a common supertype.
srsUsers.add(s1);
srsUsers.add(p);
srsUsers.add(s2);
// etc.
```



# Collections of Supertypes

- When we iterate through a collection that contains a mixture of object types, we must cast to a common supertype
  - Usually, but not always, the one furthest down on the inheritance hierarchy

```
// Iterate through the collection ...
for (int i = 0; i < srsUsers.size() ; i++) {
    Person p = (Person) srsUsers.elementAt(i);  // cast
    do things with p ...
}
```

- We'll revisit the reasons for this when we talk about polymorphism later on

# Java-Specific Collections: the ArrayList Class

- ArrayList defines many interesting behaviors:
  - add(Object) - adds an object reference to the end of the ArrayList, automatically expanding the ArrayList if need be to accommodate the reference
  - add(int, Object) - adds the specified object reference to the position in the ArrayList indicated by the int argument (where counting starts with 0, as in an Array), shifting everything in the ArrayList over by one location to accommodate the new item

# The ArrayList Class, cont.

- `set(int, Object)` - replaces the  $n^{\text{th}}$  object reference with the specified object reference
- `get(int)` - retrieves the  $n^{\text{th}}$  object reference as type `Object` - a cast is needed to 'restore' the object reference's 'true identity'
- `remove (int)` - takes out the  $n^{\text{th}}$  reference and 'closes up'/'collapses' the resultant 'hole'
- `remove(Object)` - hunts for existence of the object reference in question and, if found, removes the (first) occurrence of that reference from the `ArrayList`, closing up the 'hole'
- `size()` - returns a count of the number of `Objects`

# The ArrayList Class, cont.

- `indexOf(Object)` - hunts for existence of a specific object reference and, if found, returns an integer indicating what the (first) position is of this reference in the ArrayList (starting with 0)
  - `contains(Object)` - hunts for existence of the object reference in question and, if found, returns the value `true`, otherwise `false`
  - `isEmpty()` - returns `true` if the ArrayList is empty, `false` otherwise
  - `clear()` - empties out the ArrayList
- and there are more!

# Import Statements

- In order to use the ArrayList class in one of our own classes, we must include the appropriate import statement at the top of the enclosing class's source file:

```
// MyClass.java
import java.util.*; // ArrayList is in the java.util
// package. The import statement(s) come before the
// public class ... declaration.
public class MyClass {
    private ArrayList anArrayList;
    // etc.
```

# Import Statements

- Alternative syntax:

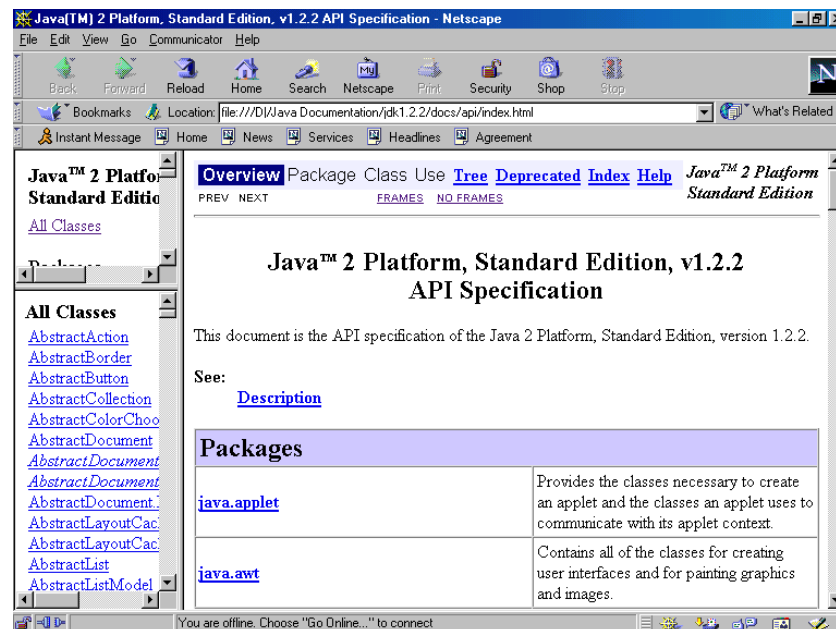
```
// MyClass.java
import java.util.ArrayList; // just import one class

public class MyClass {
    private ArrayList anArrayList;
    // etc.
```

- We'll discuss the import mechanism in more detail in a later lesson

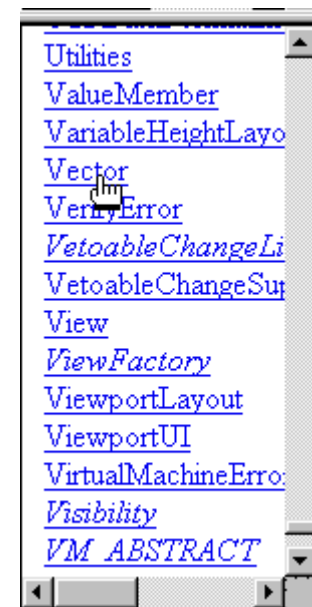
# Using On-Line Java Documentation

- Sun Microsystems has provided a convenient, on-line way to view Java language documentation from the comfort of your own browser



# On-Line Documentation, cont.

Scroll to the class of interest, and click on its hyperlink ...





# On-Line Documentation, cont.

... and documentation about that class, including its inheritance "lineage", constructors, methods, and attributes, appears in the browser window

```
java.util
Class Vector

java.lang.Object
|
+-- java.util.AbstractCollection
    |
    +-- java.util.AbstractList
        |
        +-- java.util.Vector

Direct Known Subclasses:
    Stack
```

# Representing Associations In Code, Revisited

- As mentioned earlier, we represent associations in code as attributes
  - We use individual reference variables to represent the "one" end of an association
  - We use collections to represent the "many" end of an association -- we'll illustrate this now
  - We may or may not wish to make these "bidirectional"

# Representing Associations In Code -- An Example

- A Student enrolls in many Sections, and a Section has many Students enrolled in it

```
public class Student {  
    // Attributes.  
    private String name;  
    private ArrayList sections;  
    // of Section objects  
    // etc.
```

```
public class Section {  
    // Attributes.  
    private String sectionNo;  
    private ArrayList students;  
    // of Student objects  
    Course courseRepresented;  
    // etc.
```

# Client Code for this Example

```
Student s = new Student();  
Section sec = new Section();  
// Details omitted.  
  
// Create bidirectional link between these two objects.  
sec.enroll(s);  
s.enroll(sec);
```

What would the enroll() methods look like?

# Representing Associations In Code -- Example, cont.

```
public class Student {  
    // Attributes.  
    private String name;  
    private ArrayList sections;  
    // etc.  
  
    public void enroll(Section x) {  
        // Call the add() method on the sections ArrayList.  
        sections.add(x);  
    }  
  
    // etc.
```

# Representing Associations In Code -- Example, cont.

```
public class Section {  
    // Attributes.  
    private String sectionNo;  
    private ArrayList students;  
    // etc.  
  
    public void enroll(Student x) {  
        // Call the add() method on the students ArrayList.  
        students.add(x);  
    }  
  
    // etc.
```

# Explicit Constructors

- When we have collections as attributes, we typically instantiate them in the constructor(s)
  - We never want a Student to be without a sections collection ...

```
public class Student {  
    // Attributes.  
    private String name;  
    private ArrayList sections;  
    // etc.  
  
    public void enroll(Section x) {  
        sections.add(x);  
    }  
  
    public Student() {  
        sections = new ArrayList();  
    }  
  
    public Student(String n) {  
        super(); // implied  
        name = n;  
    }  
}
```

# Explicit Constructors, cont.

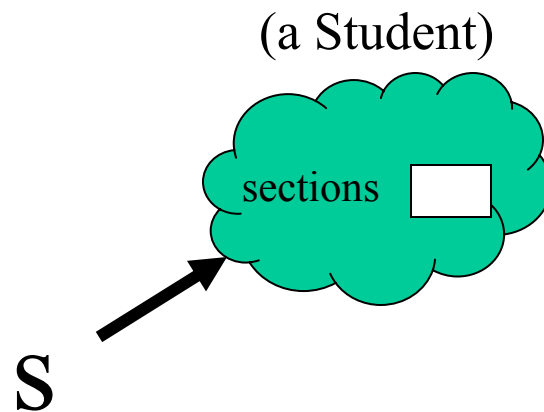
... and vice versa.

```
public class Section {  
    // Attributes.  
    private String sectionNo;  
    private ArrayList students;  
    // etc.  
  
    public void enroll(Student x) {  
        students.add(x);  
    }  
  
    public Section() {  
        students = new ArrayList();  
    }  
  
    // etc.
```



# Memory Schematic

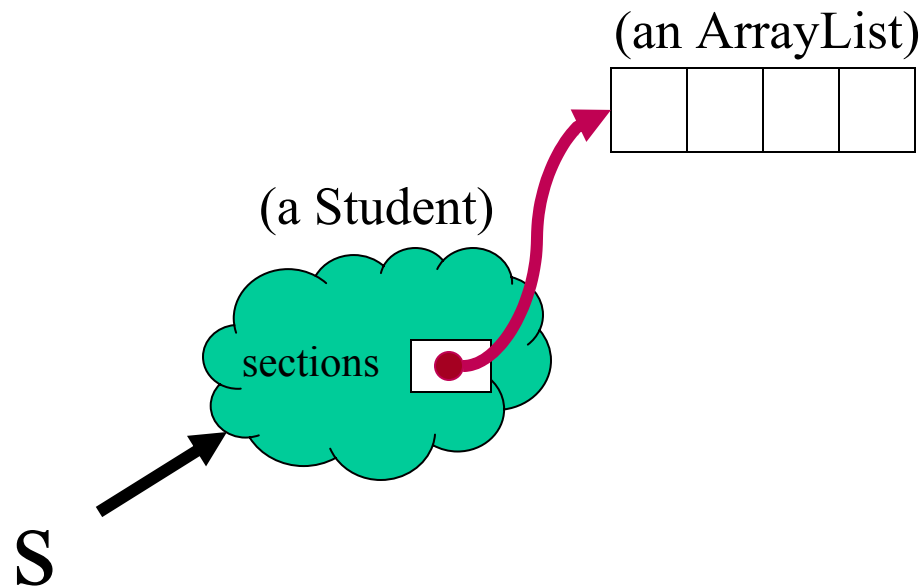
```
Student s = new Student();
```



# Memory Schematic, cont.

```
Student s = new Student();
```

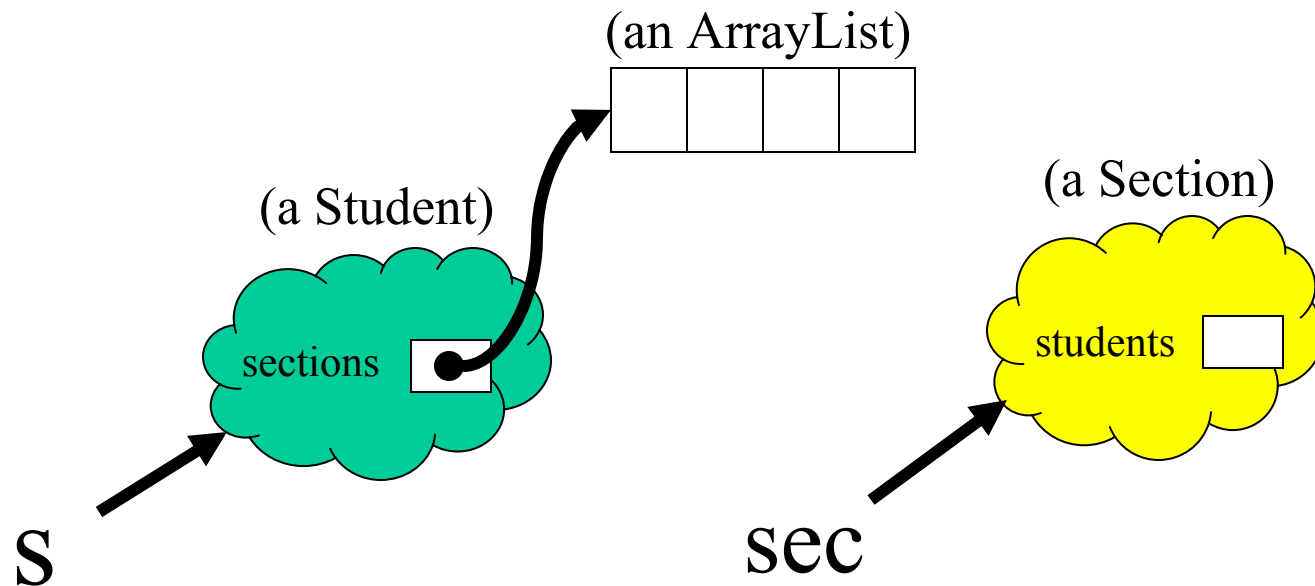
```
sections = new ArrayList();
```



# Memory Schematic, cont.

```
Student s = new Student();
```

```
Section sec = new Section();
```

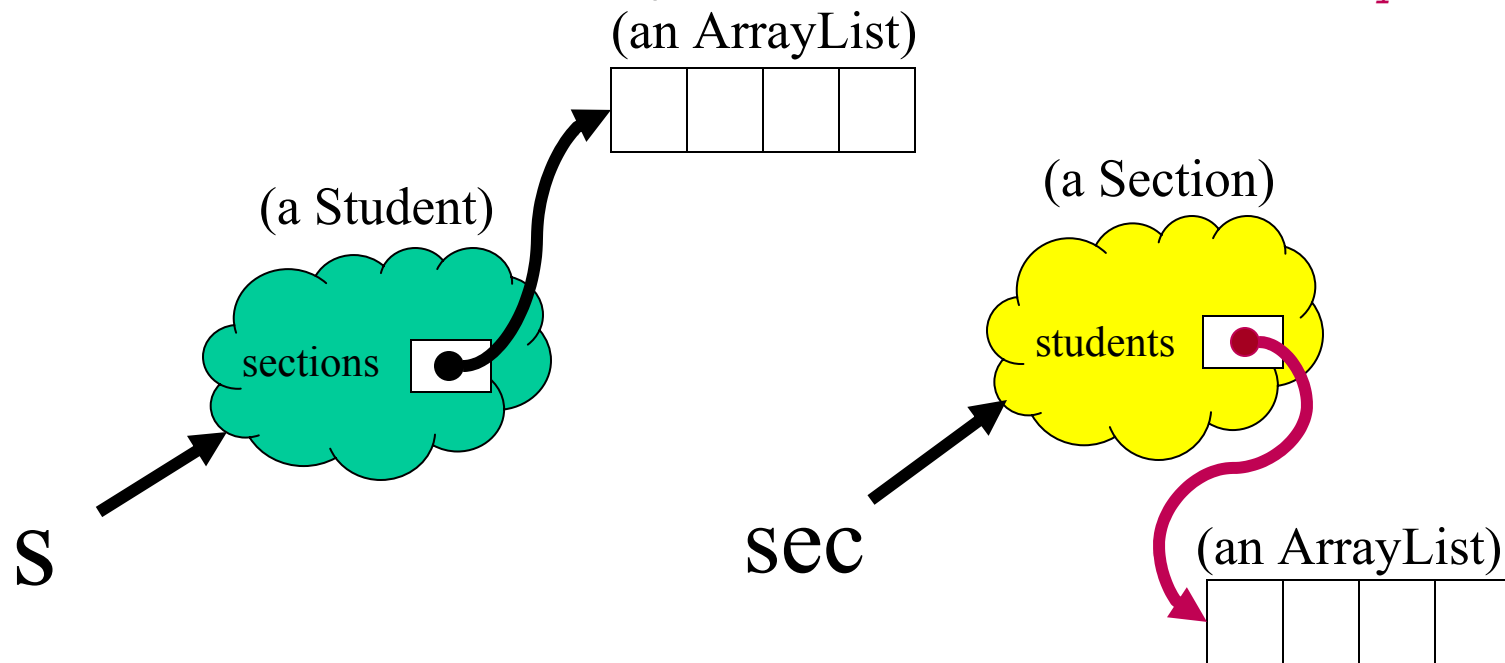


# Memory Schematic, cont.

```
Student s = new Student();
```

```
Section sec = new Section();
```

```
students = new ArrayList();
```



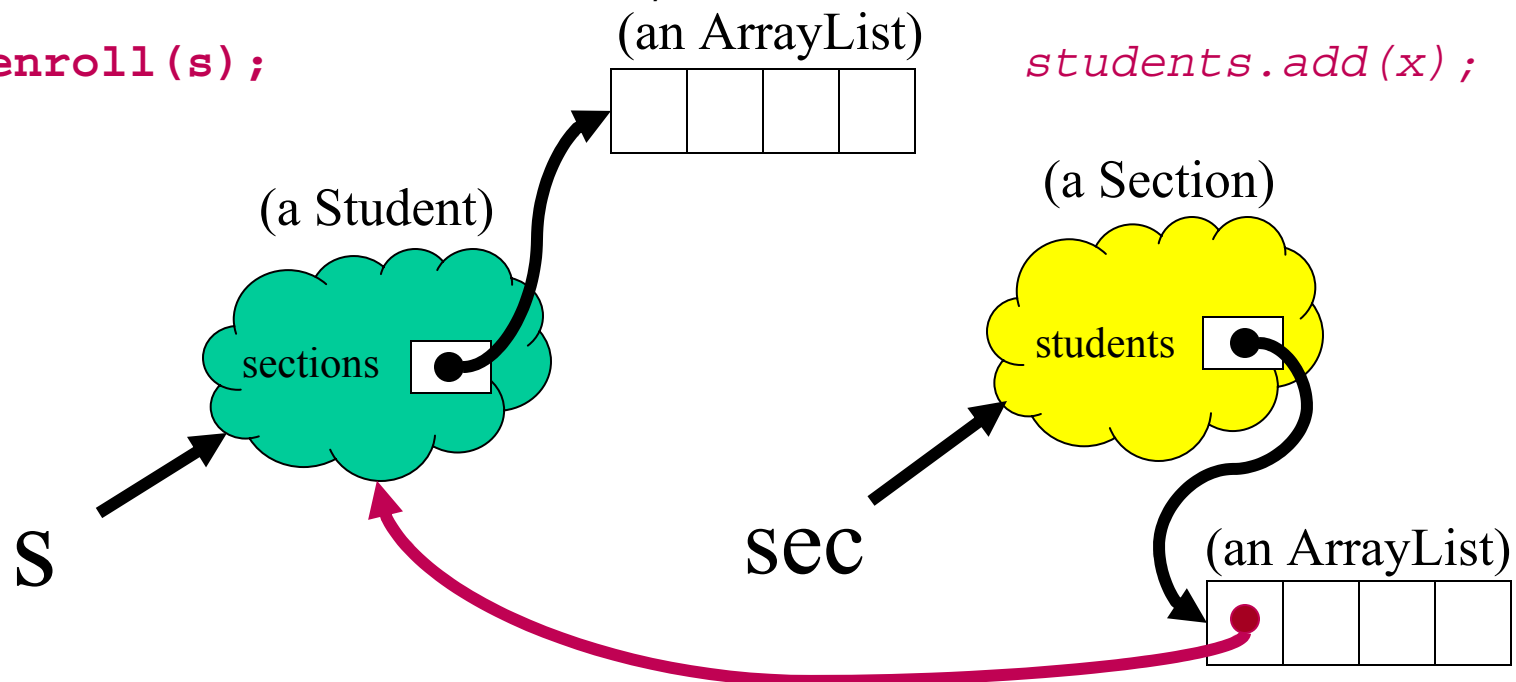
# Memory Schematic, cont.

```
Student s = new Student();
```

```
Section sec = new Section();
```

```
sec.enroll(s);
```

```
students.add(x);
```



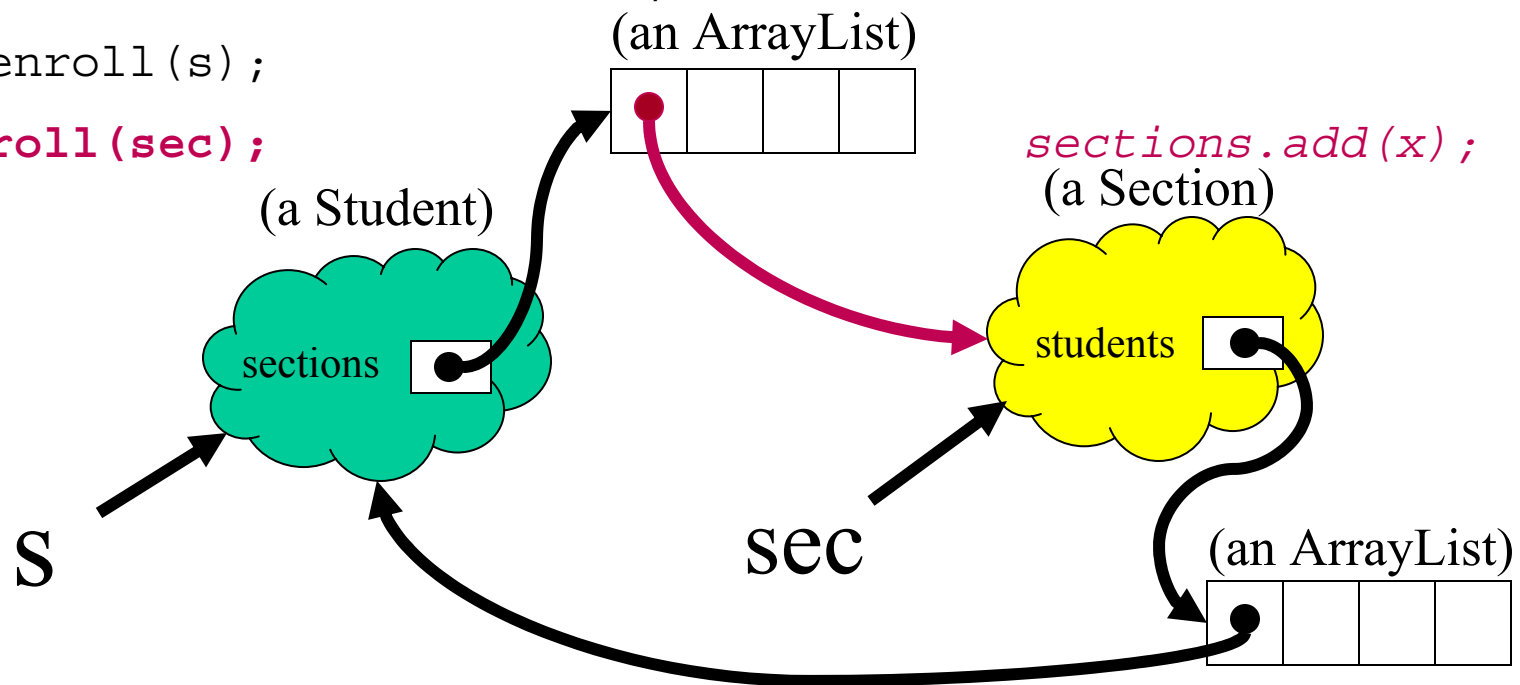
# Memory Schematic, cont.

```
Student s = new Student();
```

```
Section sec = new Section();
```

```
sec.enroll(s);
```

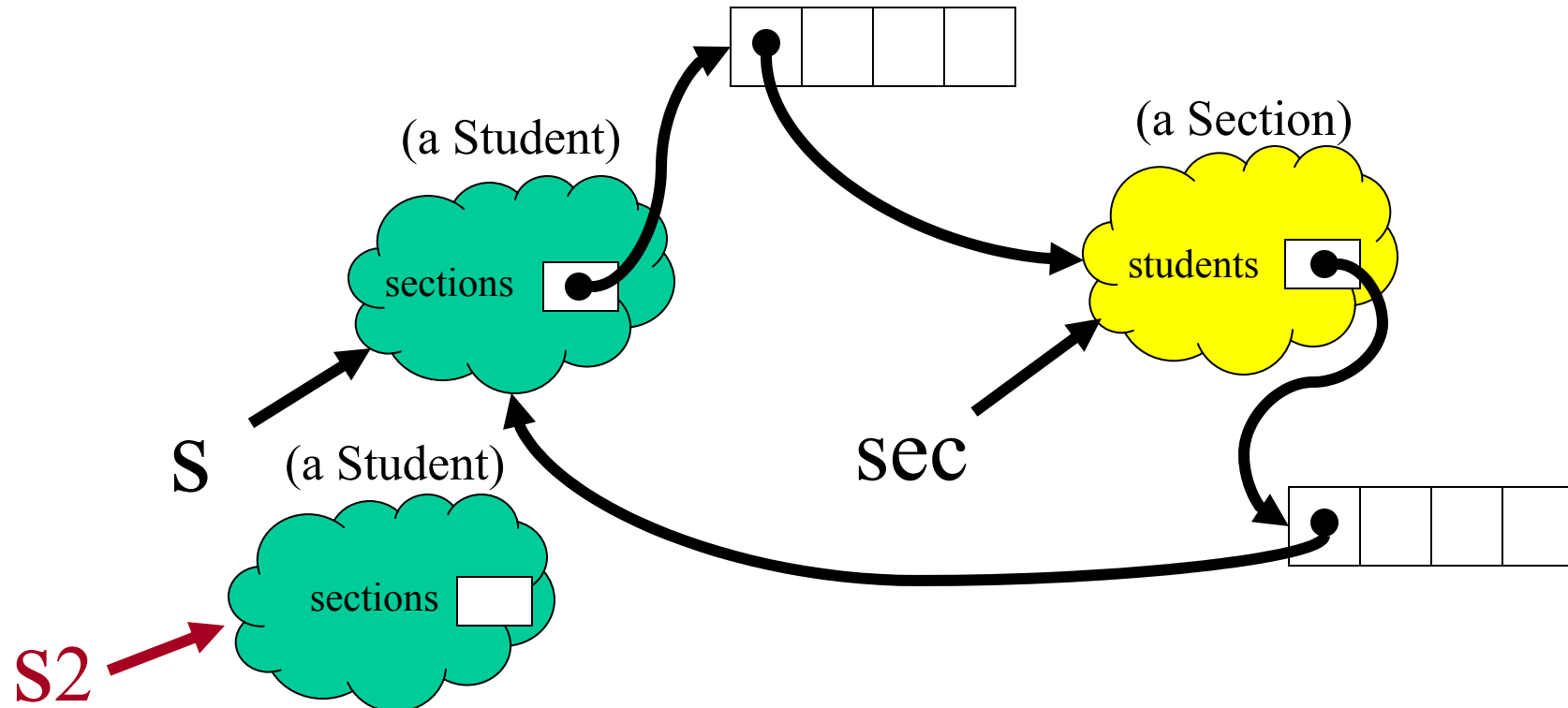
```
s.enroll(sec);
```



# Memory Schematic, cont.

```
Student s = new Student();  
Section sec = new Section();  
sec.enroll(s);  
s.enroll(sec);
```

```
Student s2 = new Student();
```

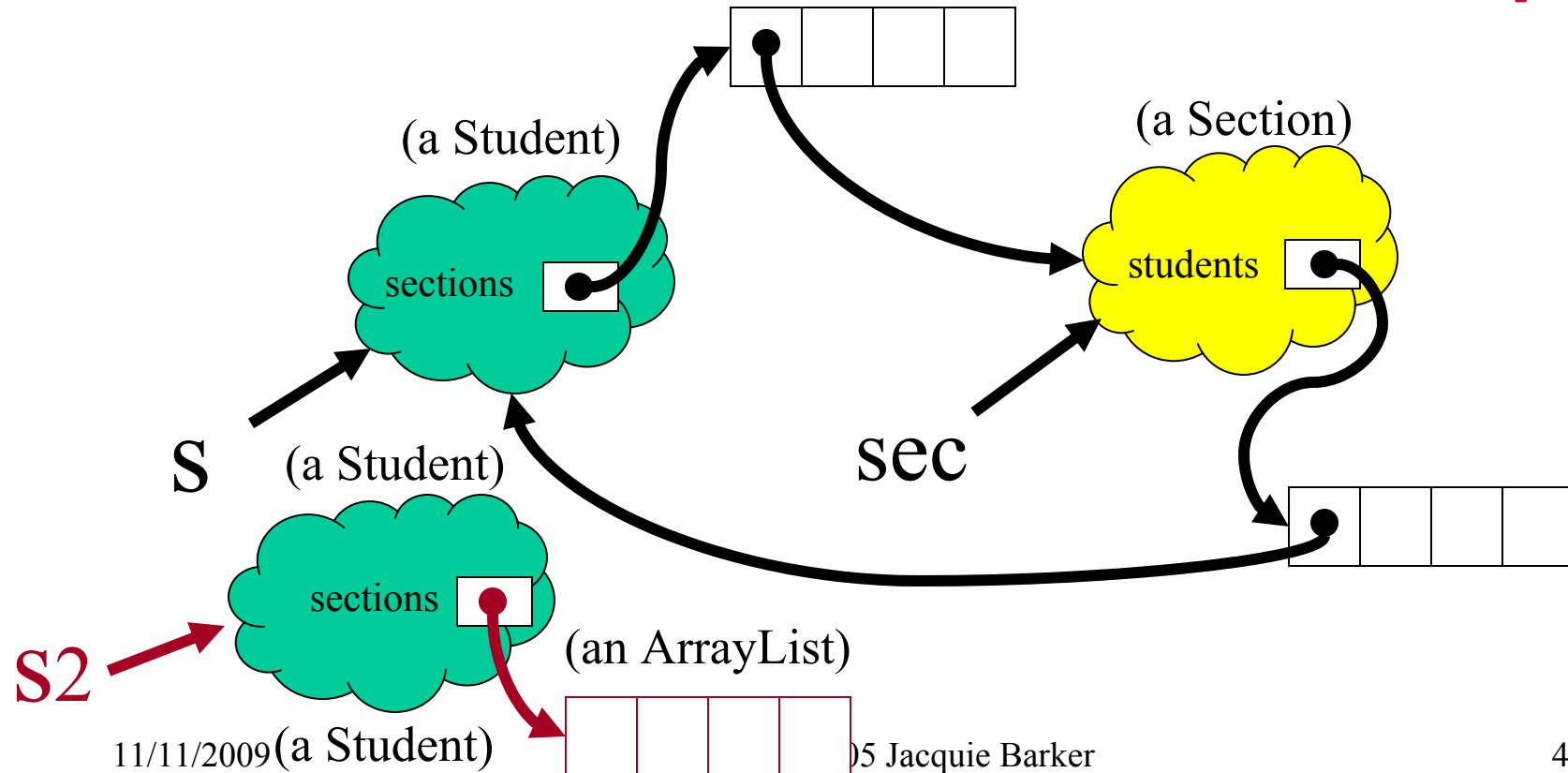


# Memory Schematic, cont.

```
Student s = new Student();  
Section sec = new Section();  
sec.enroll(s);  
s.enroll(sec);
```

```
Student s2 = new Student();
```

```
sections = new ArrayList();
```

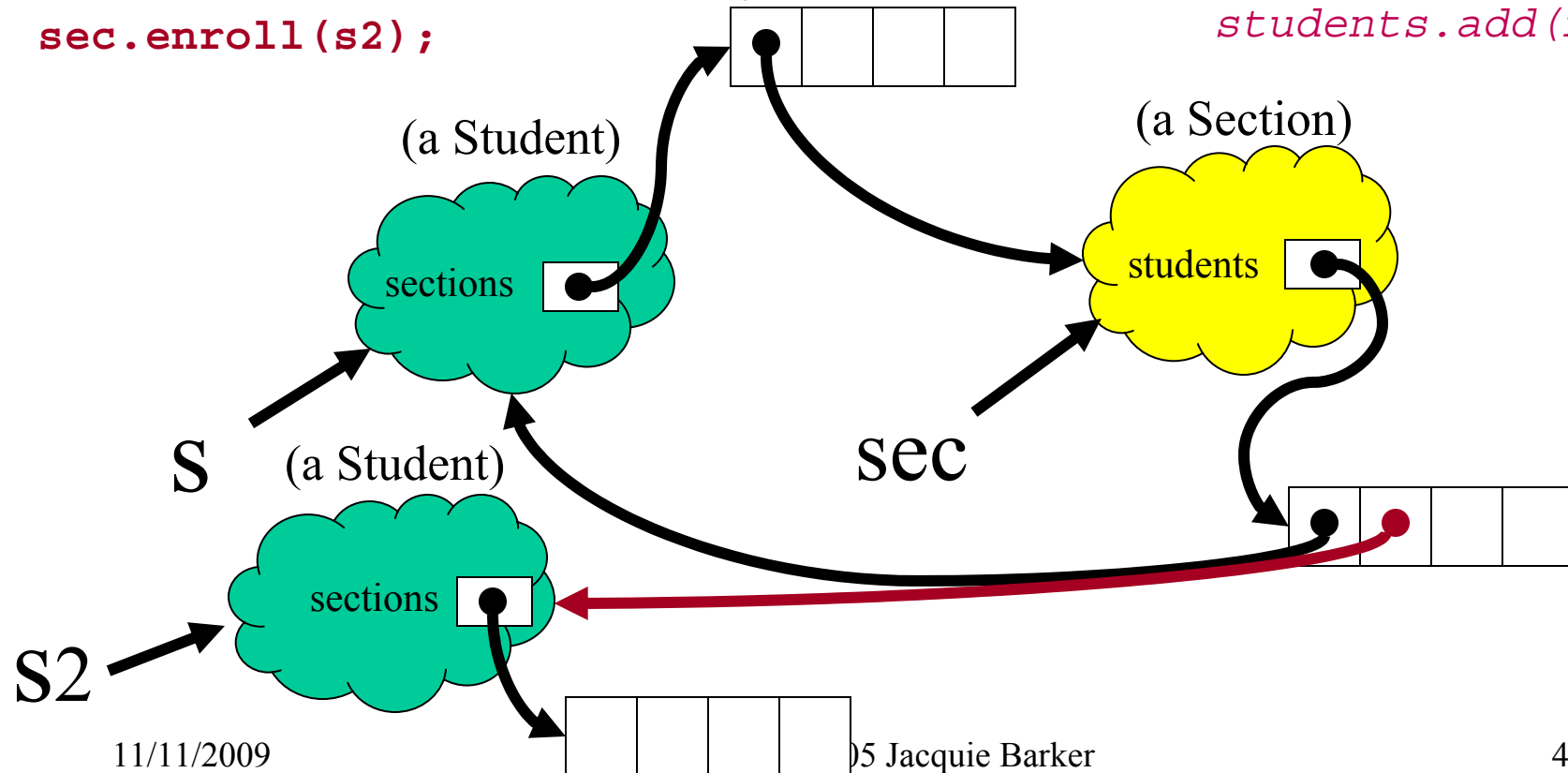




# Memory Schematic, cont.

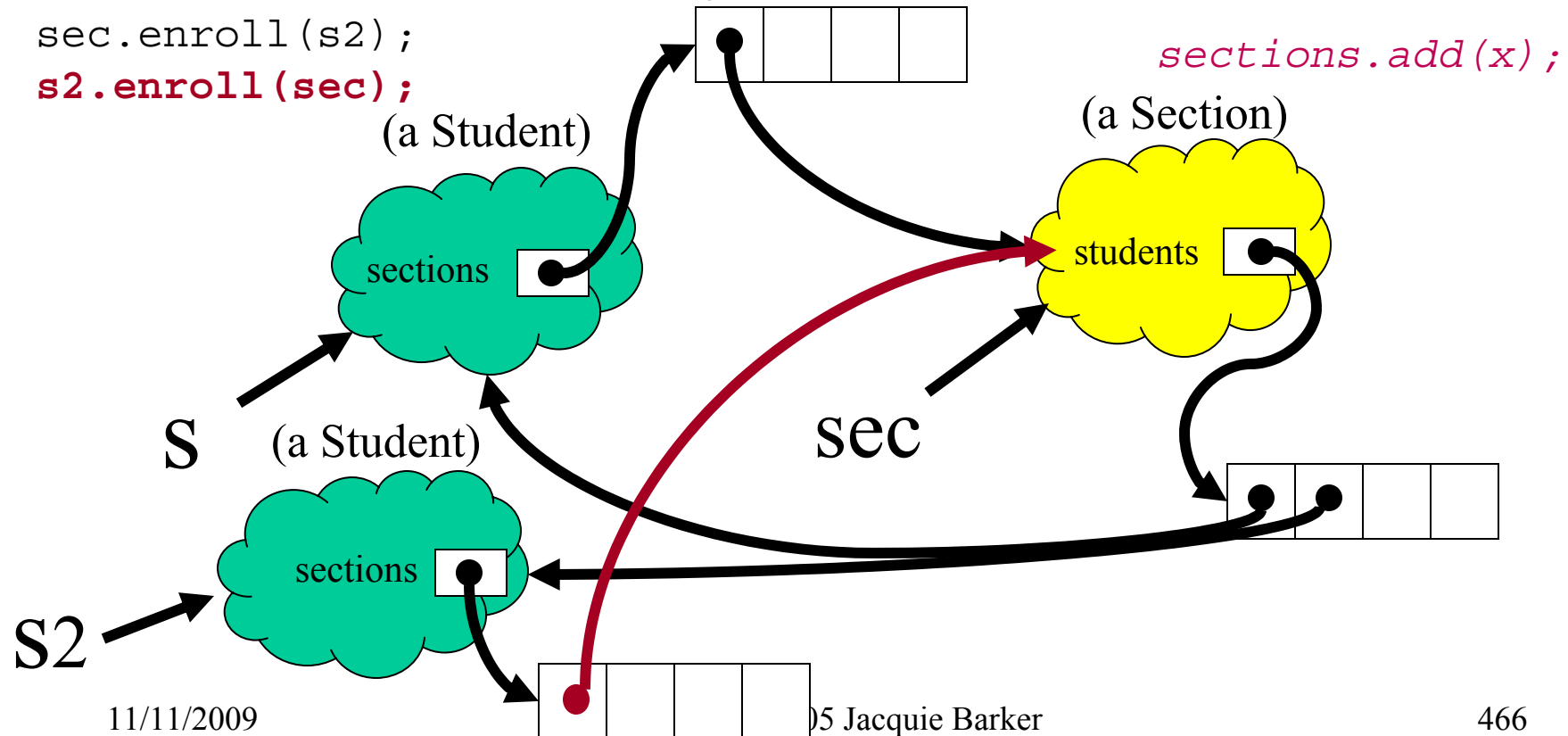
```
Student s = new Student();  
Section sec = new Section();  
sec.enroll(s);  
s.enroll(sec);  
Student s2 = new Student();  
sec.enroll(s2);
```

*students.add(x);*



# Memory Schematic, cont.

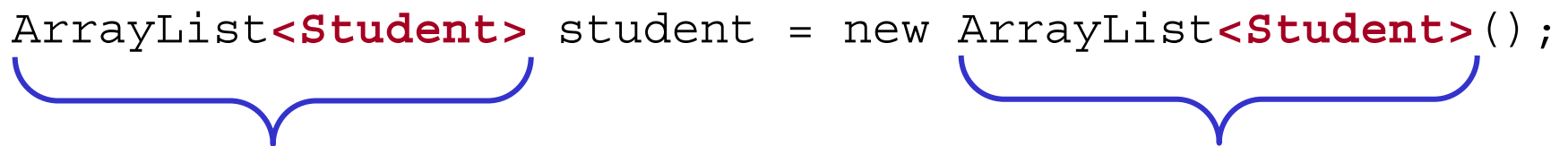
```
Student s = new Student();  
Section sec = new Section();  
sec.enroll(s);  
s.enroll(sec);  
Student s2 = new Student();  
sec.enroll(s2);  
s2.enroll(sec);
```



# Enhancements to Collections as of Java 5 (v. 1.5.x)

- With Java 1.5.x and beyond, we are able to declare collections so as to constrain their contents

```
ArrayList<Student> student = new ArrayList<Student>();
```

A blue bracket underlines the word 'Student' in the generic type 'ArrayList<Student>' on the left. Another blue bracket underlines the word 'Student' in the generic type 'ArrayList<Student>' on the right, which is part of the instantiation 'new ArrayList<Student>()'.

The collection is said to be of type  
ArrayList<Student>

# Collections as of Java 5, cont.

- We cannot insert anything but a Student (or one of its subtypes) into the collection - otherwise, the compiler will object

```
ArrayList<Student> students = new ArrayList<Student>();
```

```
Student s = new Student();
```

```
GraduateStudent gs = new GraduateStudent();
```

```
Professor p = new Professor();
```

```
students.add(s);           // allowed
```

```
students.add(gs);          // allowed
```

```
students.add(p);           // will not compile
```

# Collections as of Java 5, cont.

- The syntax for iterating through a “typed” collection is also much simpler
  - No casting required!

```
ArrayList<Student> students = new ArrayList<Student>();  
// Print a student roster.  
for (Student s : students) {  
    System.out.println(s.getName()); // No cast!  
}
```

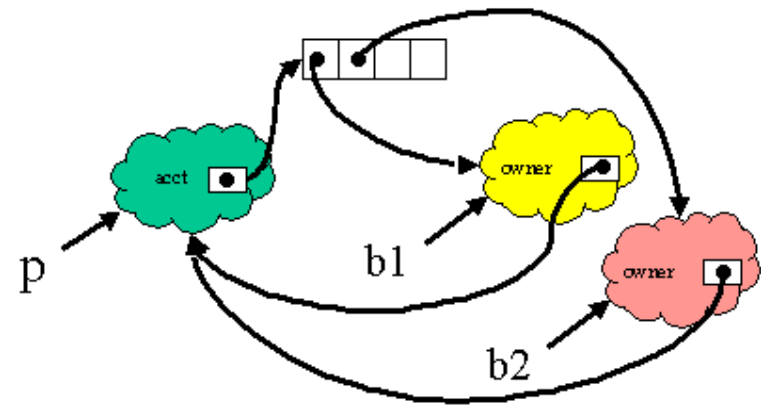
“in”

```
for (type referenceVariable : collectionName) { ... }
```

# Exercise #8

# Lab #8 Memory Schematic

- Four objects created (including the ArrayList),  
8 "handles" / references
  - From main() method:
    - p
    - b1
    - b2
  - From within Person methods:
    - acct
    - acct.elementAt(0)
    - acct.elementAt(1)
  - From within CheckingAccount methods:
    - owner
  - From within SavingsAccount methods:
    - owner



# Object Self-Referencing With 'this'

- We saw earlier that when we wish to manipulate an object, we refer to it by its reference variable in our code:

```
ArrayList x = new ArrayList();  
Student s = new Student();
```

```
// Insert the Student reference into the collection.  
x.add(s);
```



# Object Self-Referencing With 'this', cont.

- But, what do we do when we are executing the code that comprises the body of one of an object's own methods, and need the object to be able to refer to itself?

```
public class Student {  
    Professor facultyAdvisor;  
    // other details omitted  
  
    public void selectAdvisor(P professor p) {  
        // We're down in the 'bowels' of the  
        // selectAdvisor() method, executing  
        // the method for a particular object.
```

# 'this', cont.

```
// We save the handle on our new
// advisor as one of our attributes ...
facultyAdvisor = p;

// ... and now we want to turn around and
// tell this Professor object to
// add us as one of its (Student)
// advisees. The Professor class has a
// method with signature:
//     public void addAdvisee(Student s);
// so, all we need to do is call this
// method on our advisor object
// and pass in a reference to ourselves;
// but who the heck are we?
// That is, how do we refer to ourself?
p.addAdvisee(???) ;
}
```

# 'this', cont.

```
// We save the handle on our new
// advisor as one of our attributes ...
facultyAdvisor = p;

// ... and now we want to turn around and
// tell this Professor object to
// add us as one of its (Student)
// advisees. The Professor class has a
// method with signature:
//     public void addAdvisee(Student s);
// so, all we need to do is call this
// method on our advisor object
// and pass in a reference to ourselves;
// but who the heck are we?
// That is, how do we refer to ourself?
p.addAdvisee(this);  // "me"!
}
```

# 'this', cont.

- We've previously seen the use of the reserved word 'this' in two other contexts:
  - To reuse the code of one constructor from within another constructor in the same class via `this(...)`

```
public class Student {  
    // Attributes omitted - see earlier example.  
  
    // Constructor #1.  
    public Student() {  
        initialize a default student ... details omitted  
    }  
  
    // Constructor #2.  
    public Student(String s) {  
        this(); // perform code of Student()  
        do whatever extra logic makes sense  
    }  
}
```

# 'this', cont.

- As an optional dot notation prefix

```
public class SomeClass {  
    public void doOneThing() {  
        // The "this." prefix is assumed if not present.  
        this.doAnother();  
        // The above is equivalent to:    doAnother();  
    }  
  
    public void doAnother() { ... }  
}
```

# Bidirectional Linking, Revisited

- Using the 'this' keyword, we can improve our design with respect to enrolling Students in Sections

```
public class Section {  
    // Attributes.  
    private String sectionNo;  
    private ArrayList students;  
    // etc.  
  
    public void enroll(Student x) {  
        // Call the add() method on the students Vector.  
        students.add(x);  
        // Bidirectionally link the two objects.  
        x.enroll(this);  
    }  
    // etc.
```

# Client Code for this Example

```
Student s = new Student();  
Section sec = new Section();  
// Details omitted.
```

```
// Our client code is streamlined -- we only have to call  
// one method now to create a bidirectional link between  
// these two objects.
```

```
sec.enroll(s);
```

```
s.enroll(sec);
```



No longer required! The first method call does both now.

# Collections as Method Return Types

- We may overcome the limitation that a method can only return one object by returning a collection object

```
class Section {  
    String sectionNo;  
    ArrayList students;  
    // etc.  
  
    ArrayList getStudents() {  
        return students;  
    }  
  
    // etc.
```



# Collection Return Types, cont.

- Client code:

```
// Enroll TWO students in a section.
sec.enroll(s1);
sec.enroll(s2);  // etc.

// Now, ask the section to give us a handle on the
// collection of all of its registered students ...
ArrayList students = sec.getStudents();

// ... and iterate through the collection, printing
// out a grade report for each Student.
for (int i = 0; i < students.size(); i++) {
    Student s = (Student) students.elementAt(i);
    s.printGradeReport();
}
```

# Composite Classes, Revisited

- When we first looked at the attributes of the Student class, we were stumped on a few types

Attribute Name	Data Type
name	String
studentID	String
birthdate	Date
address	String
major	String
gpa	float
advisor	Professor
<b>courseLoad</b>	<b>??? ←</b>
<b>transcript</b>	<b>??? ←</b>

- We can now address these using collections

## Composite Classes, cont.

- The `courseLoad` attribute can be declared to be simply a collection of `Course` objects
  - Perhaps an `ArrayList`
- The `transcript` attribute is a bit more complex
  - What is a transcript, in 'real world' terms?
  - A list of all of the courses that a student has taken, along with the semester in which each course was taken and the letter grade that the student received for the course

# Composite Classes, cont.

- If we invent a TranscriptEntry class:

```
public class TranscriptEntry {
    private Course courseTaken;
    private String semesterTaken; // "Spring 2000"
    private String gradeReceived; // e.g., "B+"

    // Get/set methods omitted from this example ...

    public void printTranscriptEntry() {
        // Reminder:  "\t" is a tab character,
        // "\n" is a newline.
        System.out.println(semesterTaken + "\t" +
            courseTaken.getCourseNo() + "\t" +
            courseTaken.getTitle() + "\t" +
            courseTaken.getCreditHours() + "\t" +
            gradeReceived);
    }
}
```

# Composite Classes, cont.

... then a Transcript can simply be represented as a collection of TranscriptEntry objects:

```
public class Student {
    private String name;
    private String studentId;
    private ArrayList transcript; // of TranscriptEntry objects

    public void addTranscriptEntry(TranscriptEntry te) {
        transcript.add(te);
    }

    public void printTranscript(String filename) {
        for (int i = 0; i < transcript.size(); i++) {
            TranscriptEntry te =
                (TranscriptEntry) transcript.elementAt(i);
            te.printTranscriptEntry();
        }
    }
    // etc.
```

# Composite Classes, cont.

- We could even construct the TranscriptEntry object on the fly, internally to the Student class:

```
class Student {
    String name;
    String studentId;
    ArrayList transcript; // of TranscriptEntry objects

    public void completedCourse(Course c, String semester,
                               String grade) {
        // (This assumes we've written the appropriate
        // TranscriptEntry constructor.)
        TranscriptEntry te = new TranscriptEntry(c, semester,
                                                grade);

        transcript.add(te);
    }

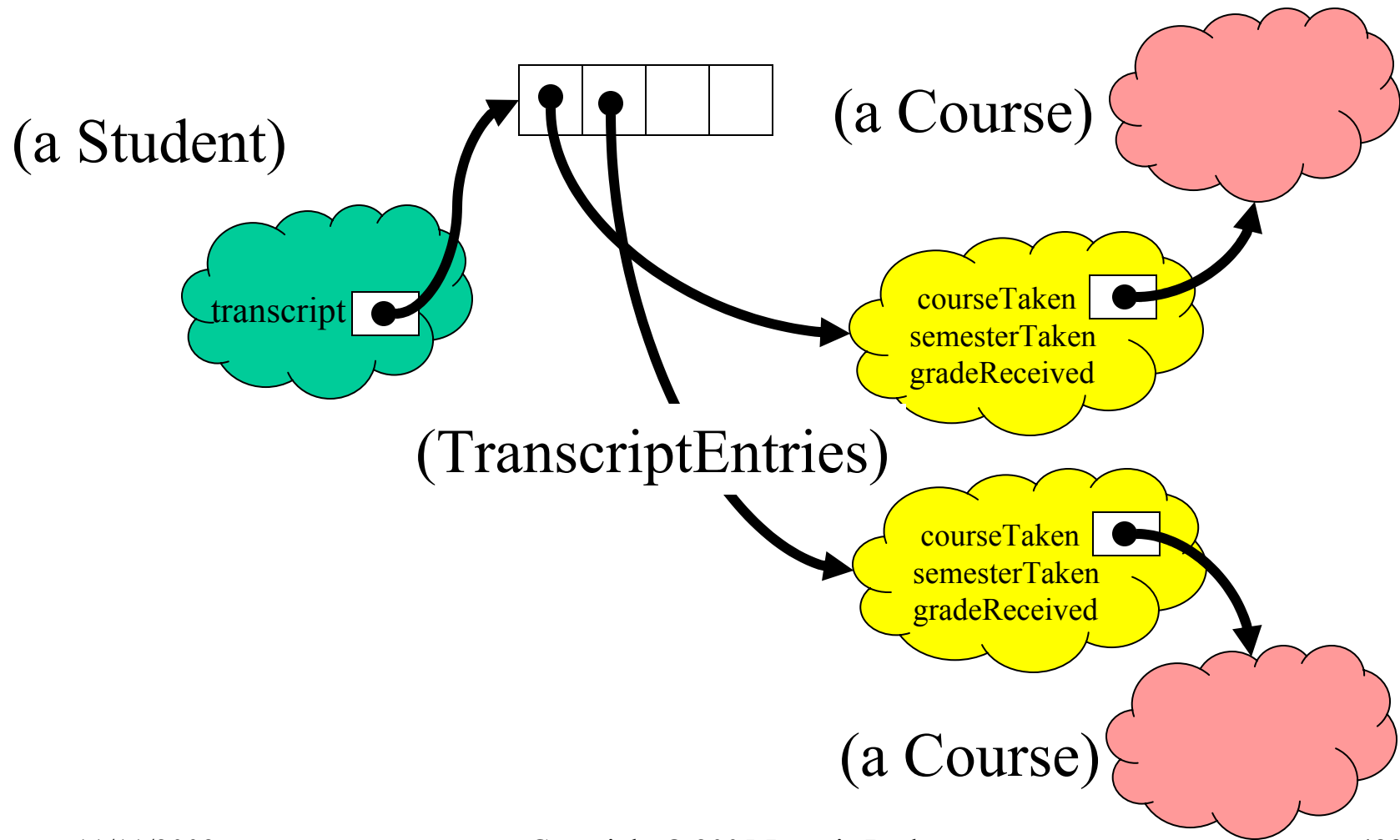
    // etc.
```

# Composite Classes, cont.

- Sample client code for this example:

```
public static void main(String[] args) {  
    Student s = new Student();  
    Course c = new Course();  
    // details omitted  
  
    s.completedCourse(c, "FALL 2002", "A");  
    s.printTranscript("transcript.dat");  
  
    // etc.
```

# Memory Schematic





# Composite Classes, cont.

- Even better, we could create a Transcript class as an abstraction, to encapsulate all of the logic for creating/manipulating TranscriptEntry objects ...

```
public class Transcript {  
    ArrayList transcriptEntries; // of TranscriptEntry objects  
  
    public void addTranscriptEntry(Course c, String semester,  
                                   String grade) {  
        TranscriptEntry te = new TranscriptEntry(c, semester,  
                                                  grade);  
        transcript.add(te);  
    }  
  
    public void printTranscript(String filename) {  
        for (int i = 0; i < transcript.size(); i++) {  
            TranscriptEntry te =  
                (TranscriptEntry) transcript.elementAt(i);  
            te.printTranscriptEntry();  
        }  
    }  
}
```

11/11/2009  
}

# Composite Classes, cont.

- ... which would greatly simplify our Student class!

```
public class Student {  
    String name;  
    Transcript transcript;  
  
    public Student() {  
        transcript = new Transcript();  
    }  
  
    public void completedCourse(Course c, String semester,  
                               String grade) {  
        // Delegation!  
        transcript.addTranscriptEntry(c, semester, grade);  
    }  
  
    public void printTranscript(String filename) {  
        // Delegation!  
        transcript.printTranscript(filename);  
    }  
}
```

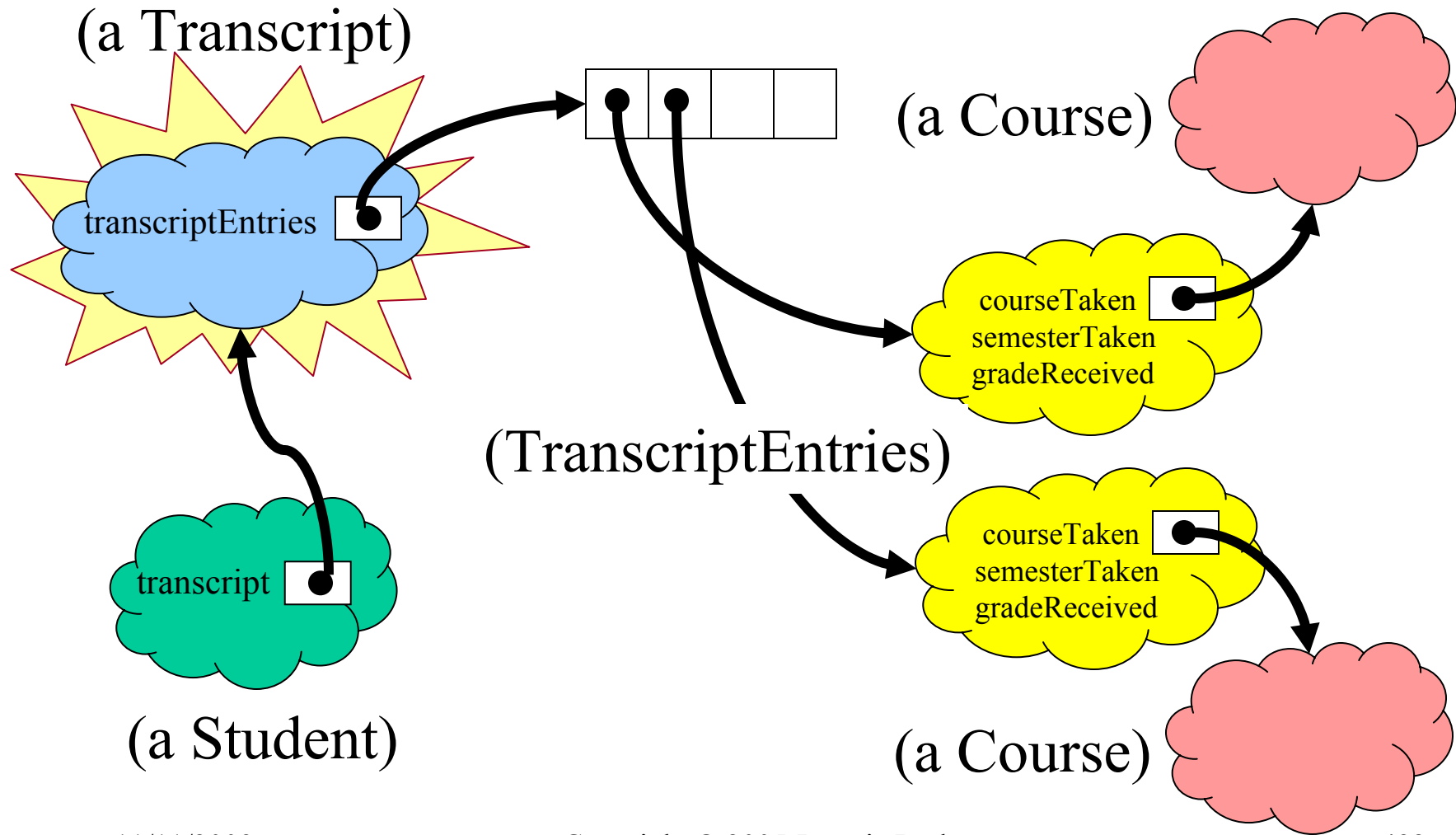
11/11/2009

# Composite Classes, cont.

- Sample client code for this example is identical:

```
public static void main(String[] args) {  
    Student s = new Student();  
    Course c = new Course();  
    // details omitted  
  
    s.completedCourse(c, "FALL 2002", "A");  
    s.printTranscript("transcript.dat");  
  
    // etc.
```

# Memory Schematic



# Object Modeling

- The decision of which abstractions/classes to create, and how to relate these together, occurs during the object modeling process

# Composite Classes, Revisited

- Our completed Student class data structure:

Attribute Name	Data Type
name	String
studentID	String
birthdate	Date
address	String
major	String
gpa	float
advisor	Professor
<b>course Load</b>	ArrayList
<b>transcript</b>	ArrayList (or Transcript)

# Object Concept Wrap-Up

## (Chapter 7)



*Polymorphism ...  
abstract classes ...  
interfaces ...  
static features ...*

# Polymorphism

- The ability of two or more objects belonging to different classes to respond to exactly the same message in different class-specific ways
  - A surgeon, a hair stylist, and an actor
- Let's create an array called `studentBody` declared to hold references to `Student` objects (continued on next slide)



# Polymorphism, cont.

```
// Declare and instantiate an array.
Student[] studentBody = new Student[20];

// Instantiate various types of Student object.
UndergraduateStudent u1 = new UndergraduateStudent();
UndergraduateStudent u2 = new UndergraduateStudent();
GraduateStudent g1 = new GraduateStudent();
GraduateStudent g2 = new GraduateStudent();

// 'Stuff' them into the array in random order.
// (This is permitted by virtue of the "is a" relationship.)
studentBody[0] = u1;
studentBody[1] = g1;
studentBody[2] = g2;
studentBody[3] = u2;
```

# Polymorphism, cont.

- Assume that the print() method of Student is overridden in its various subclasses

```
// Step through the array (collection) ...  
for (int i = 0; i < studentBody.length; i++) {  
    // ... invoking the print() method of the ith  
    // student object.  
    studentBody[i].print();  
}
```

- We might see the output on the following page:

# Polymorphism, cont.

Student Name: John Smith  
Student No.: 12345  
Major Field: Biology  
GPA: 2.7

High School Attended: James Ford Rhodes High  
THIS IS AN UNDERGRADUATE STUDENT ...

Student Name: Paula Green  
Student No.: 34567  
Major Field: Education  
GPA: 3.6

Undergrad. Deg.: B.S. English  
Undergrad. Inst.: UCLA  
THIS IS A GRADUATE STUDENT ...

(etc.)

# Polymorphism, cont.

- The term **polymorphism** is defined in Webster's dictionary as:
  - *'The quality or state of being able to assume different forms'.*
- The line of code:

```
studentBody[i].print();
```

is said to be **polymorphic** because the method code performed in response to the message can take many different forms, depending on the class identity of the object

# Polymorphism, cont.

- Another example, based on our lab work:
  - Create an ArrayList of BankAccount objects -- a random mixture of CheckingAccounts and SavingsAccounts
  - Iterate through the collection, asking each one to perform the same service:

```
for (BankAccount b : bankAccounts) {  
    b.display(); // polymorphic  
}
```

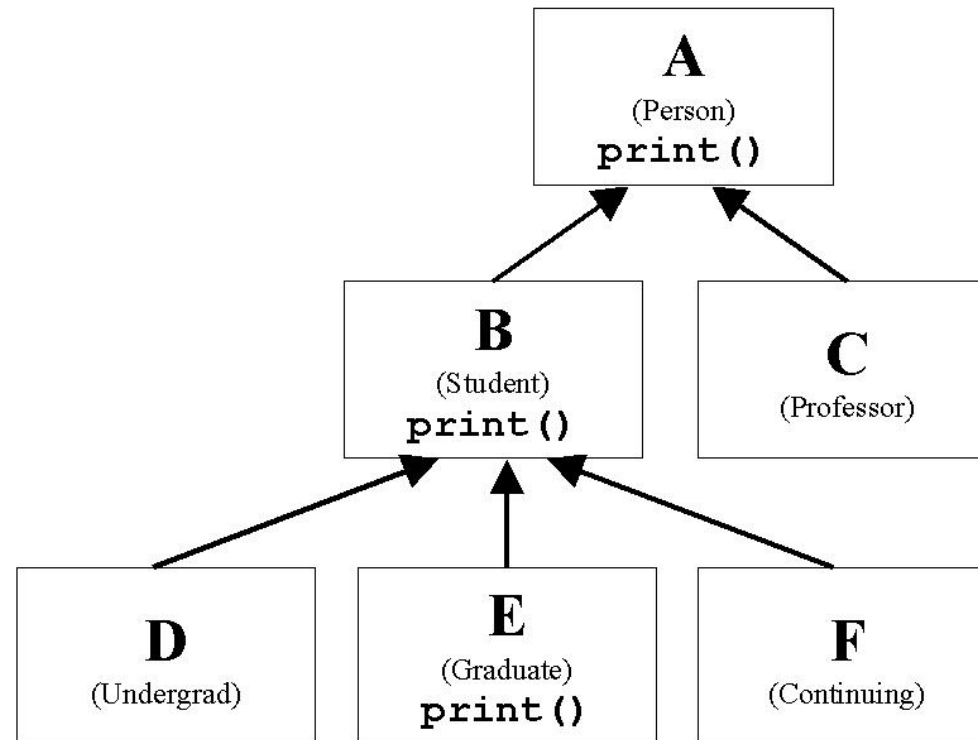
# Polymorphism, cont.

- We see that we already knew everything that we needed to know about Java objects to enable polymorphism
  - Inheritance
  - Overriding
- Polymorphism doesn't happen automatically in C++
  - Virtual functions

# Polymorphism, cont.

- Iterating through a collection to ask objects to each do something in its own class-specific way won't work unless all objects in the collection understand the message being sent
- We constrained the array to only hold Student objects (or subclasses thereof), so we are *guaranteed* that they will all know what to do when asked to `print()`
  - By virtue of inheritance, any subclass of Student will have either:
    - Inherited the Student's version of the `print()` method, or
    - Overridden it with one of its own

# Polymorphism, cont.



We cannot "uninherit" the `print()` method!



# Reminder

- As we learned earlier, had we chosen a different Java collection type, then **prior to Java 1.5.x** we would not have been able to constrain the type of Objects to be inserted when we declared the collection
- So, we had to exercise programming discipline when inserting objects into such a collection to ensure that they all speak a common language in terms of messages that they understand
  - The compiler won't stop you from putting objects IN
  - The compiler/JVM WILL complain when you try to operate on the objects after taking them back OUT

# Pre-1.5.x Example

```
Vector v = new Vector();

// Assume that Object is the only common superclass of
// the next three objects' classes.
Pineapple p = new Pineapple();
Bicycle b = new Bicycle();
Cloud c = new Cloud();

// Add to the vector.
v.add(p);
v.add(b);
v.add(c);

// So far, so good! Now, let's try to pull them out.
```

## Pre 1.5.x Example, cont.

```
for (int i = 0; i < 3; i++) {  
    // No casting needed -- Object is the default, and is the  
    // common superclass of all of the objects.  
    Object o = v.elementAt(i);  
    o.someMethod();    // Can only be a method defined by the  
                       // Object class (e.g., toString());  
                       // otherwise, the compiler will object,  
                       // because all it has to go by is the  
                       // type of "o".  
}
```

---

or, alternatively:

```
for (int i = 0; i < 3; i++) {  
    // Will generate a runtime ClassCastException on non-  
    // Pineapple objects.  
    Pineapple p = (Pineapple) v.elementAt(i);  
    p.eat();    // Compiler is happy if eat() is defined  
               // as a method on the Pineapple class.  
}
```

# Versus 1.5.x and Beyond

```
// Since we are able to constrain the type of element to be  
// inserted into a collection as of Java 5 ...  
Vector<type> v = new Vector<type>();
```

```
// ... we are PREVENTED from mixing unlike objects in such  
// a collection.
```

```
Pineapple p = new Pineapple();  
Bicycle b = new Bicycle();  
Cloud c = new Cloud();
```

```
// These next three lines won't all compile unless <type>  
// above is <Object>.  
v.add(p);  
v.add(b);  
v.add(c);
```

# Polymorphism Simplifies Code Maintenance

- To appreciate the power of polymorphism, let's look at how we might have to approach the challenge of handling different objects in different type-specific ways with a non-OO programming language
  - We'd typically handle all of the various scenarios having to do with different kinds of students using a series of 'if' tests (see next page)

# Simplified Maintenance, cont.

```
for (int i = 0; i < studentBody.size(); i++) {  
    // Process the ith student.  
    Student s = (Student) studentBody.elementAt(i);  
    // (pseudocode)  
    if (s is an undergraduate student)  
        printAsUndergraduateStudent(s);  
    else if (s is a graduate student)  
        printAsGraduateStudent(s);  
    else if ...  
}
```

- As the number of cases grows, so too does the 'spaghetti' nature of the resultant code!

# Simplified Maintenance, cont.

- With a polymorphic language, we can invent as many subtypes after the fact as we like -- e.g., GraduateStudent => MastersStudent, PhDStudent -- and the client code that manipulates these needn't change!

```
// SAME CODE AS BEFORE!  
// Step through the array (collection) ...  
for (int i = 0; i < studentBody.size(); i++) {  
    // ... invoking the print() method of the ith  
    // student object.  
    Student s = (Student) studentBody.elementAt(i);  
    s.print(); // Works for ALL types of Student!  
}
```

# Three Features of an OOPL

- ✓ (Programmer creation of) Abstract Data Types (ADTs)
- ✓ Inheritance
- ✓ Polymorphism



# Abstract Classes/Methods

- Say that we have the foresight to know that we are going to need various types of Course objects in our SRS
  - Lecture courses, lab courses, independent study courses, etc.
- We want to design the Course (super)class to be as versatile as possible to facilitate future specialization
- All Courses, regardless of type, are going to need to share a few common attributes and behaviors (cont.)

# Abstract Classes, cont.

- Common attributes:

```
String courseName  
String courseNumber  
int creditValue  
Vector enrolledStudents  
Professor instructor
```

- Some of the behaviors may be generic enough to program in detail for the Course class, knowing that any future subclasses of Course will most likely inherit these methods 'as is' without needing to override them

```
enrollStudent()  
assignProfessor()
```

# Abstract Classes, cont.

- Others may need to be specialized: e.g.,  
`establishCourseSchedule()`
  - A lecture course may only meet once a week for 3 hours;
  - A lab course may meet twice a week for 2 hours each time;
  - An independent study course may meet on a custom schedule agreed to by a given student and professor
- It would be a waste of time for us to bother trying to program a generic version of this method
- Yet, we *know* that we'll need such a method for all subclasses of `Course` that get created down the road
- How do we communicate the requirement for such a behavior in all subclasses of `Course` and, more importantly, *enforce its future implementation?*

# Abstract Classes, cont.

- OOPLs support the notion of an **abstract class**
- Used to enumerate the required behaviors of a class without having to provide an explicit implementation of each and every such behavior
  - For those behaviors for which we cannot (or care not to) devise a generic implementation, we are allowed to specify method *signatures* without having to program the method *bodies*
  - We refer to a 'codeless', or signature-only, method specification as an **abstract method**
  - An abstract class may also contain normal methods with code bodies (aka **concrete methods**)

# Abstract Classes, cont.

```
public abstract class Course {  
    String courseNumber;  
    Vector enrolledStudents;  
    Professor instructor;  
    // etc.  
  
    public boolean enrollStudent(Student s) {  
        if (maximum enrollment not exceeded) {  
            enrolledStudents.add(s);  
            return true;  
        }  
        else return false;  
    }  
  
    public abstract void establishCourseSchedule(  
        String startDate, String endDate);  
  
    // etc.
```

# Abstract Classes and Inheritance

- If we derive a subclass from an abstract class, we must override all of its abstract methods to provide concrete method bodies if we wish to "break the spell" of abstractness

```
public class IndependentStudyCourse extends Course {  
    // Details (other attributes/methods) omitted ...  
    // Override all abstract methods w/ concrete methods.  
    public void establishCourseSchedule(  
        String startDate, String endDate) {  
        provide a code body ...  
    }  
    // etc.
```

"abstract"  
keyword  
removed



# Abstract Classes and Inheritance, cont.

- If we fail to provide concrete method bodies for all of the abstract methods that we've inherited:
  - They are inherited as abstract methods, and
  - The compiler will force us to declare the subclass as an abstract class, as well

# Abstract Classes, cont.

- By specifying an abstract method, we:
  - Specify a service that objects belonging to/descended from this class must be able to perform (i.e., mandate requirements)
  - Detail the means by which we will ask them to perform this service by providing a method signature (message format)
  - Facilitate polymorphism by ensuring that all subclasses of *Course* will indeed recognize such a message

but without pinning down the private details of how the method will accomplish this behavior ('what' without 'how')



# Abstract Classes and Instantiation

- We cannot instantiate an abstract class:

```
Course c = new Course(); // Impossible!
```

class Course is an abstract class. It can't be instantiated.

- This makes intuitive sense, because there are service(s) that such an object could not perform

```
c.establishCourseSchedule("01/10/2001", "05/15/2001");  
// Behavior undefined!
```

# Abstract Classes and Variable Declaration

- We are allowed to declare references belonging to an abstract class, however

```
Course c; // Fine and dandy!
```

- This enables us to take advantage of polymorphism, e.g., when iterating through a collection

```
Vector courseCatalog; // of Courses
// Populate the collection (details omitted) ...
// Step through the collection.
for (int i = 0; i < courseCatalog.size(); i++) {
    Course c = (Course) courseCatalog.elementAt(i);
    c.establishCourseSchedule("01/10/2001",
        "05/15/2001"); // polymorphic
}
```

# Exercise #9

# Abstractness

- Recall that a class, as an abstract data type, is an abstraction of a real world object
- We can see that an abstract class is 'more abstract' than a 'normal' class because we've omitted the details for how one or more particular behaviors are to be performed
- But, with an abstract class, we typically still specify the data structure (attributes)
- What if we wanted to specify *only* abstract behaviors?

# Abstractness, cont.

- That is, we may not wish to constrain what (private) attributes/data structure a future class must use in order to achieve a desired (public) behavior
- Say, for example, that we wanted to define what it means to teach at a university -- services:
  - Agree to teach a particular course
  - Designate a textbook to be used for the course
  - Define a syllabus for the course
  - Approve the enrollment of a particular student in the course.

# Abstractness, cont.

- Each of these behaviors could be formalized with a method signature, representing how an object that is *capable of teaching* would be asked to perform them:

```
public void agreeToTeach(Course c)
```

```
public void designateTextbook(TextBook b, Course c)
```

```
public Syllabus defineSyllabus(Course c)
```

```
public boolean approveEnrollment(Student s, Course c)
```

# Abstractness, cont.

- We could certainly declare an abstract class that declares only abstract methods without declaring any attributes or concrete methods:

```
public abstract class Teacher {  
    // NO data structure!  
    // All methods abstract!  
    public abstract void agreeToTeach(Course c);  
    public abstract void designateTextbook(TextBook b, Course c);  
    public abstract Syllabus defineSyllabus(Course c)  
    public abstract boolean approveEnrollment(Student s, Course c);  
}
```

- We'd then create concrete subclasses of Teacher, which would add private attributes and override public methods

# Interfaces

- The preferred approach is to declare a Java **interface**, which is a set of (abstract) method signatures that collectively define what it means to assume a certain role (such as teaching)

`// Teacher.java`

```
public interface Teacher {  
    // Note: no "abstract" keyword required.  
    public void agreeToTeach(Course c);  
    public void designateTextbook(Book b, Course c);  
    public Syllabus defineSyllabus(Course c);  
    public boolean approveEnrollment(Student s, Course c);  
}
```



# Interfaces, cont.

- We then indicate that a particular class performs these behaviors by implementing the interface and overriding the methods (just as we did with abstract classes):

```
public class Professor implements Teacher {  
    // Desired attributes go here ...  
  
    // Override each method signature.  
    public void agreeToTeach(Course c) {  
        code body provided ...  
    }  
  
    public void designateTextbook(Book b, Course c) {  
        code body provided ...  
    }  
}
```

11/11/2009

Copyright © 2005 Jacquie Barker

529

// and so on

# Interfaces, cont.

- In terms of the 'abstractness spectrum', an interface is even more abstract than an abstract class, which is in turn more abstract than a 'regular' class
- A class may be instructed to implement as many interfaces as desired (by contrast, a class may extend only one parent class)
- For example, if we were to invent a second interface called 'Administrator', which specified the following method signatures:

```
public boolean approveNewCourse(Course c);  
public boolean hireProfessor(Professor p);
```

# Interfaces, cont.

then we could instruct a class to implement both the Teacher and Administrator interfaces:

```
class Professor implements Teacher, Administrator { ... }
```

When a class implements more than one interface, it effectively assumes multiple identities or roles

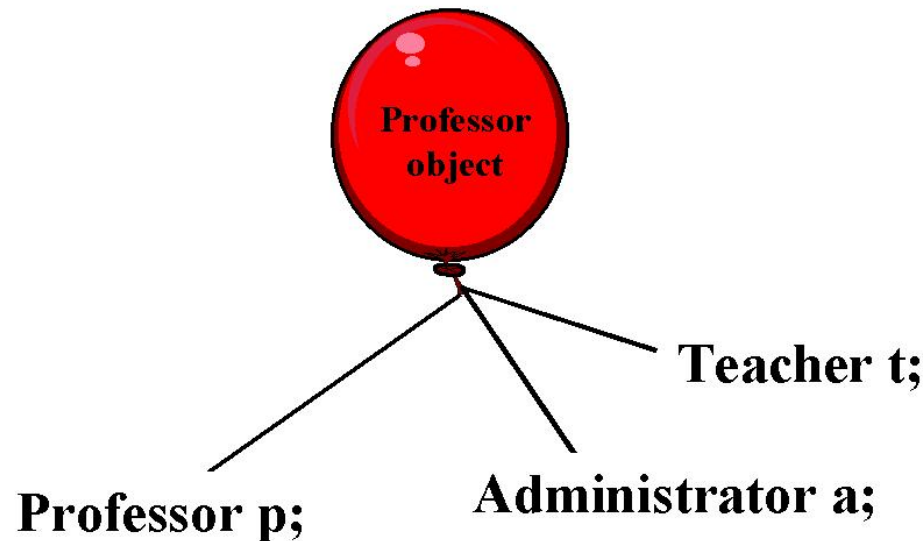
- Its 'handle' can therefore be managed by various types of reference variables

# Interfaces, cont.

```
// We instantiate a Professor object, and store its
// 'handle' in a reference variable of type Professor.
Professor p = new Professor();

// We declare two references of types defined by
// interfaces.
Teacher t;
Administrator a;
t = p; // We store a 'handle' on the Professor in a
      // reference variable of type Teacher
a = p; // We store a 'handle' on the Professor in a
      // reference variable of type Administrator
```

# Interfaces, cont.



*A Professor object can be “handled” by Professor, Teacher, and Administrator references, because a Professor is a Teacher, and a Professor is an Administrator!*

- An interface is therefore another way of implementing the "is a" relationship!

# Interfaces, cont.

- We may then command the same object as either a Professor ...

```
p.getName();
```

or as a Teacher ...

```
t.agreeToTeach(c);    // p.agreeToTeach(c); also works ...
```

or as an Administrator ...

```
a.approveNewCourse(c);    // p.approveNewCourse(c) also  
                           // works ...
```

because it is all three!

# Interfaces, cont.

- We cannot do the following, however:

```
t.getName();           // getName() is defined for the Professor
                        // class, but not all Teachers are
                        // Professors
```

because even though, in our scenario, `t` is referring to a Professor object at runtime, the compiler will object: not all Teachers are Professors

# Interfaces, cont.

- If we also wish to enable Students to be teachers:

```
class Professor implements Teacher, Administrator { ... }  
class Student implements Teacher { ... }
```

we may then use Student and Professor object references interchangeably wherever a Teacher object reference is required



# Versatility of Interfaces

```
public class Course {  
    Teacher instructor;    // This is a more versatile design.  
  
    public void setInstructor(Teacher t) {  
        instructor = t;  
    }  
    // etc.
```

---

```
public static void main(String[] args) {  
    Student s = new Student();  
    Professor p = new Professor();  
    Course c = new Course();  
  
    c.setInstructor(p);    -or-    c.setInstructor(s);
```

Either will work just fine!

# Interfaces, cont.

- An interesting example of casting to consider:

```
Professor p = new Professor(); // implements Teacher
Student s = new Student(); // implements Teacher
Teacher t;
```

```
t = p; // This is fine, because a Professor is a Teacher.
p = t; // How about this? Will it compile/run properly?
```

```
}
```

```
}
```

# Interfaces, cont.

- An interesting example of casting to consider:

```
Professor p = new Professor(); // implements Teacher
Student s = new Student(); // implements Teacher
Teacher t;
```

```
t = p; // This is fine.
```

```
p = t; // WON'T COMPILE! Not all Teachers are Professors.
```

```
// (Remember, the compiler is guessing at compile time
```

```
// what types of objects the references will refer to
```

```
// at runtime.)
```

```
// What might we do to fix the above?
```

```
}
```

```
}
```

# Interfaces, cont.

- An interesting example of casting to consider:

```
Professor p = new Professor(); // implements Teacher
Student s = new Student(); // implements Teacher
Teacher t;
```

```
t = p; // This is fine.
```

```
p = (Professor) t; // Will compile and run fine now.
```

```
}
```

```
}
```

# Interfaces, cont.

- An interesting example of casting to consider:

```
Professor p = new Professor(); // implements Teacher
Student s = new Student(); // implements Teacher
Teacher t;
```

```
t = p; // This is fine.
```

```
p = (Professor) t; // Will compile and will run fine.
```

```
t = s; // t now refers to our Student object.
```

```
p = (Professor) t; // Will this compile/run properly?
```

```
}
```

```
}
```

# Interfaces, cont.

- An interesting example of casting to consider:

```
Professor p = new Professor(); // implements Teacher
Student s = new Student(); // implements Teacher
Teacher t;
```

```
t = p; // This is fine.
```

```
p = (Professor) t; // Will compile and will run fine.
```

```
t = s; // t now refers to a Student.
```

```
p = (Professor) t; // Will compile, but will throw a
                  // runtime ClassCastException.
```

```
}
```

```
}
```

# Interfaces vs. Multiple Inheritance

- It is argued that interfaces make up for the lack of multiple inheritance in Java
  - As we've seen, it is possible to create objects that are hybrids of two different concepts
  - ProfessorStudent could be a class that implements two interfaces: Teaches and Studies
- This makes up for the *behavioral* side of multiple inheritance
- There isn't a simple solution, however, as to how to handle the need for hybrids of the *data structures* of two different entities

# Interfaces vs. Multiple Inheritance, cont.

- It can be argued that, since the data structure is usually private, it is less essential than the public behaviors when creating hybrid classes/objects
- Nonetheless, it is desirable to reuse code, which includes attribute declarations, whenever possible
- There are object modeling techniques for making up for the lack of multiple inheritance in a language
  - See Object-Oriented Modeling and Design by James Rumbaugh et al, Prentice Hall, 1991; Chapter 4, Section 4.4 "MULTIPLE INHERITANCE", section 4.4.3 "Workarounds"



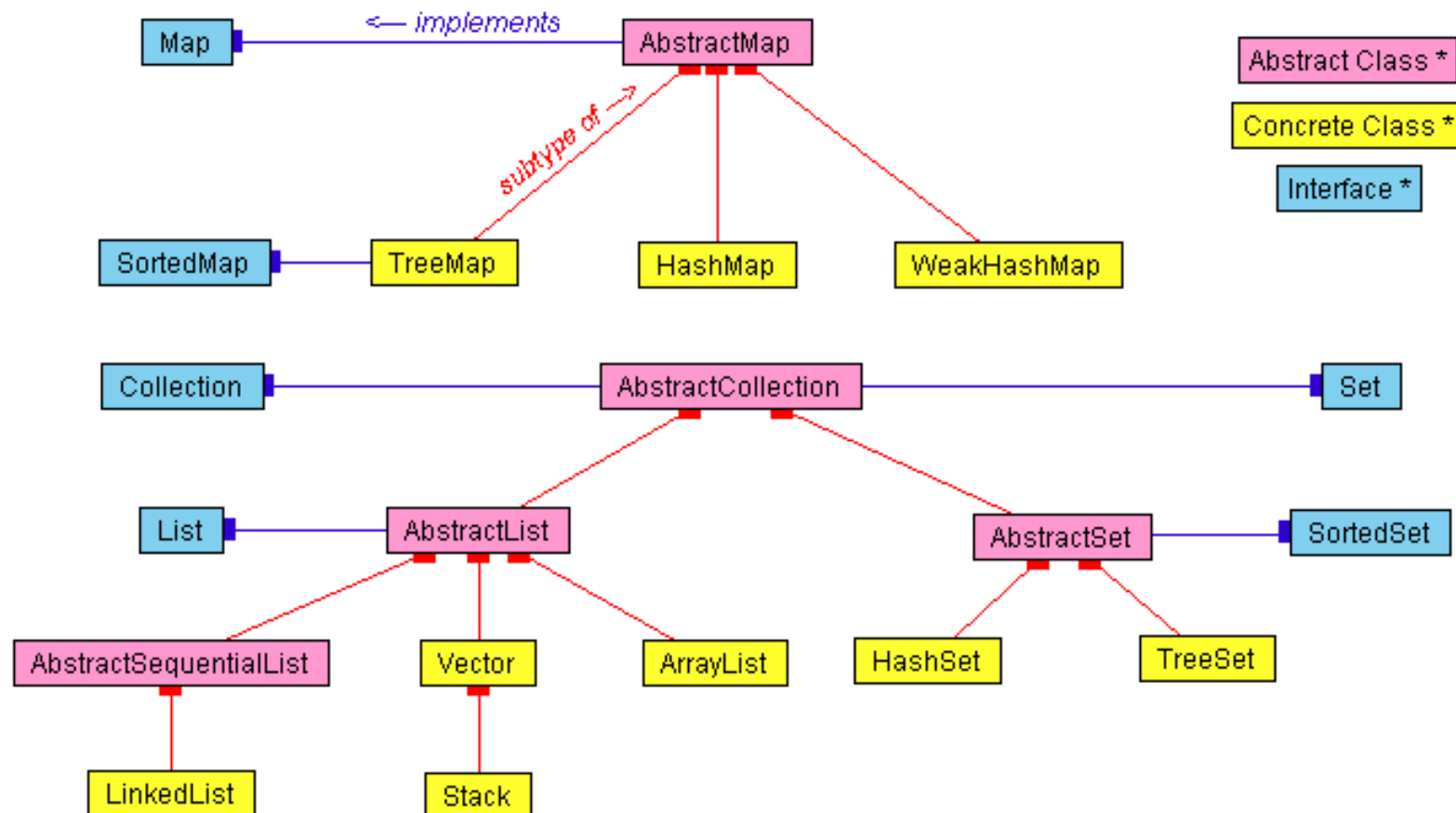
# Interfaces, cont.

- An example of a commonly used built-in interface: the Java Collection interface
  - Enforces implementation of many method signatures

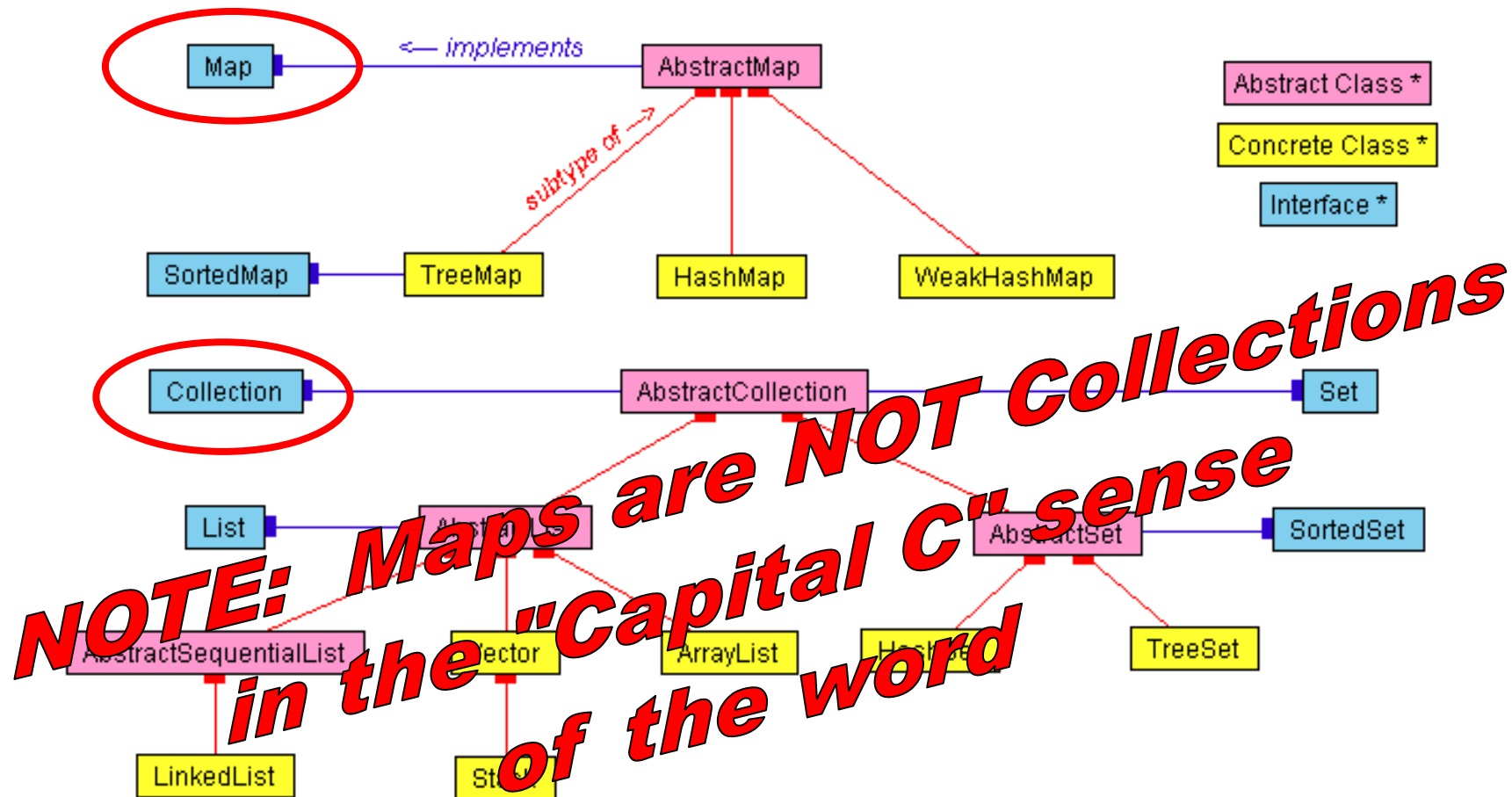
```
boolean add(Object o)
boolean addAll(Collection c)
void clear()
boolean contains(Object o)
boolean containsAll(Collection c)
boolean isEmpty()
boolean remove(Object o)
boolean removeAll(Collection c)
int size()
etc. (15 in all)
```

- Implemented by many built in classes (see next slide)

# Interfaces, cont.



# Interfaces, cont.



# Versatility of Interfaces

- Whenever possible/feasible, design the public aspects of your classes:

- arguments to methods
- method return types


using interface types instead of specific class types to allow for greater flexibility/utility of your methods

- An example follows ...

# Versatility of Interfaces

- Example: by returning a generic Collection instead of an ArrayList, we hide the fact that we're using an ArrayList, and can switch collection types later on

```
public class Section {  
    private String sectionNo;  
    private ArrayList enrolledStudents;  
    // etc.  
  
    Collection getEnrolledStudents() {  
        // An ArrayList is a Collection!  
        return enrolledStudents;  
    }  
    // etc.
```



# Interfaces and Inheritance

- Interfaces can also be related via inheritance

```
public interface Set extends Collection { ... }  
public interface SortedSet extends Set { ... }
```

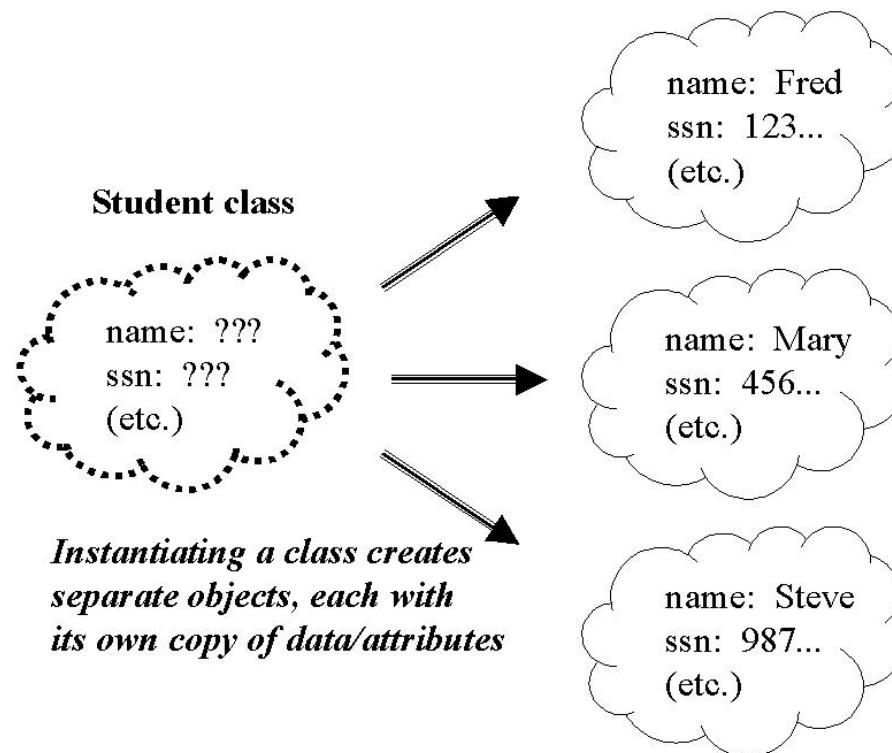
- Finally, a given class may extend another class and implement multiple interfaces

```
public class Professor extends Person implements Teacher,  
    Administrator { ... }
```

# Exercise #10

# Static Attributes

- When we create an object, we are creating an instance of the appropriate class 'template' which gets filled in with attribute values



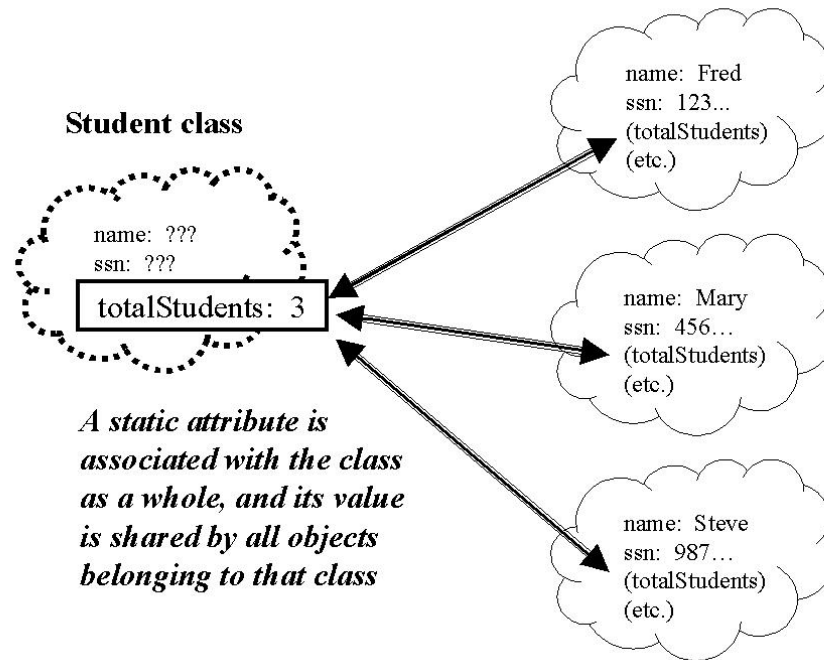


## Static Attributes, cont.

- Suppose there were some piece of general information - say, the total student enrollment count at the university - that we wanted *all* Student objects to have shared access to
  - We could implement this as a 'standard' attribute of the Student class -- say, `int totalStudents ...`
  - ... but then we'd have to work hard to make sure that all of the different Student objects' values for `totalStudents` were in synch

# Static Attributes, cont.

- A **static attribute** is one whose value is shared by all instances of that class



- It in essence belongs to the class as a whole
- "Global" to all instances of that class

# Static Attributes, cont.

```
class Student {
    private String name;
    // Note that we may initialize static attr. as we
    // declare them. They'll get initialized the
    // first time we reference this class in our app.

    private static int totalStudents = 0;
    // details omitted ...

    // Constructor.
    public Student(String n) {
        setName(n);
        // Increment the student count.
        totalStudents++;
    }

    public int getTotalStudents() {
        return totalStudents;
    }
}
```

# Static Attributes, cont.

```
// Client code:
```

```
Student s1 = new Student("Fred");  
Student s2 = new Student("Mary");  
Student s3 = new Student("Steve");
```

```
// All of these method calls will return the value 3.  
s1.getTotalStudents();  
s2.getTotalStudents();  
s3.getTotalStudents();
```

# Static Attributes, cont.

```
// Client code:
```

```
Student s1 = new Student("Fred");  
Student s2 = new Student("Mary");  
Student s3 = new Student("Steve");
```

```
// All of these method calls will return the value 3.  
s1.getTotalStudents();  
s2.getTotalStudents();  
s3.getTotalStudents();
```

```
// So will THIS!  
Student.getTotalStudents();
```

# Accessing Static Attributes

- The latter of these messages illustrates what's known as a **static method**: i.e., one which can be invoked on the class as a whole

```
public class Student {  
    private String name;  
    private static int totalStudents = 0;  
    // other attribute details omitted ...  
  
    public static int getTotalStudents() {  
        return totalStudents;  
    }  
  
    // etc.
```

# Static Methods

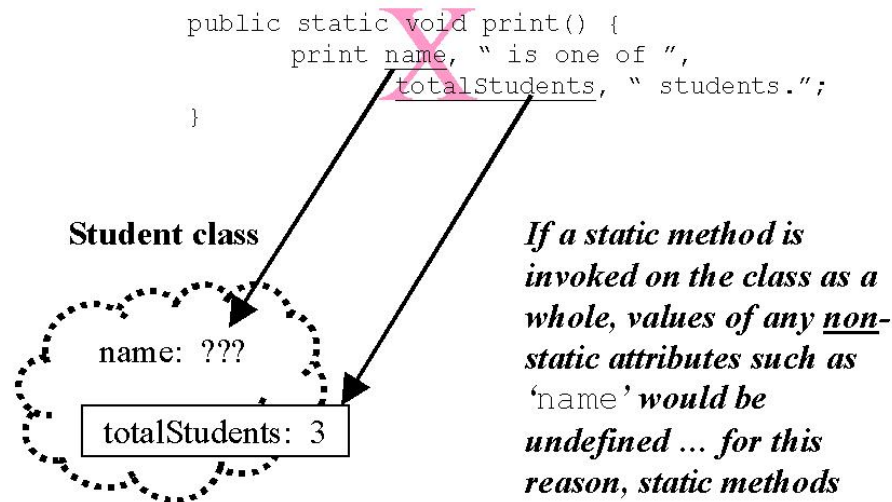
- Important restriction on static methods: they may *only* access *static* attributes:

```
class Student {  
    private String name; // NOT static  
    private static int totalStudents;  
    public static void print() {  
        System.out.println(name + " is one of " +  
            totalStudents + " students."); // ILLEGAL!  
    }  
}
```

Can't make a static reference to nonstatic variable name in class Student

# Static Methods, cont.

- Why is this?
  - Classes are for the most part empty templates
  - If a static method is invoked on a class as a whole, and tries to access a non-static attribute, the value of that attribute would be undefined





# Exercise #11

# The Static Nature of main()

- Why must the main() method of an application be declared to be static?

```
public static void main(String[] args) { ...
```

- To answer this question, let's review what happens when we first launch an application

```
java MyApp
```

- the JVM is looking for a class file with the specified name, loading it into its memory, and looking for/running the main() method

# The Nature of main(), cont.

```
java MyApp
```

- Step 1: the JVM looks for a class file with the specified name
- Step 2: if found, it is loaded into the JVM's memory
- Step 3: the JVM inspects the class for the existence of a main() method with the appropriate signature
- Step 4: if found, the main() method is executed -- but, we haven't created have any objects yet, so we must look to the class to run this method => it must be a static method!

# Utility Classes

- Some classes consist predominantly/wholly of static methods (and sometimes public static final (constant) attributes), and are created as a convenient way of bundling utility functions
  - Example: `java.lang.Math`
  - Defines static methods for performing basic numeric operations along with elementary exponential, logarithm, square root, and trigonometric functions
  - We never instantiate a `Math` object -- instead, we use the methods statically on the `Math` class as a whole, as a sort of math "library"

# Math Class Example

```
public class MathExample {  
    public static void main(String[] args) {  
        // We never instantiate the Math class!  
        double d = -3.4;  
        double dabs = Math.abs(d); // absolute value  
        double radius = 3.7;  
        // A public static (constant) ATTRIBUTE!  
        double circumference = 2.0 * Math.PI * radius;  
        double area = Math.PI * Math.pow(radius, 2.0);  
        double e = 7.92;  
        // Compute the square root of d squared plus e squared.  
        double answer = Math.sqrt(Math.pow(d, 2.0) +  
                                   Math.pow(e, 2.0)); // nested  
        // etc.  
    }  
}
```

11/11/2009

# Custom Utility Classes

- We can take advantage of this technique to write our own utility classes

```
public class TempUtility {  
    // Define a public, constant (unchanging) attribute.  
    public static final double CENTIGRADE_BOILING = 100.0;  
    public static final double FAHRENHEIT_BOILING = 212.0;  
  
    public static double centigradeToFahrenheit(double c) {  
        return (c * (9.0/5.0)) + 32.0;  
    }  
  
    public static double fahrenheitToCentigrade(double f) {  
        return (f - 32.0) * (5.0/9.0);  
    }  
  
    // etc.
```

# Custom Utility Classes

- Sample client code:

```
public static void main(String[] args) {  
    Soup s = new Soup("Chicken Noodle");  
    s.cook(5);  
    while (s.getTemp() < TempUtility.CENTIGRADE_BOILING) {  
        System.out.println("Soup's temperature is " +  
            TempUtility.centigradeToFahrenheit(s.getTemp()));  
        s.cook(1);  
    }  
  
    // etc.
```

# Alternative Terminology

- To differentiate between 'normal' (non-static) attributes and static attributes:
  - '**instance variable**' is often used for the former: namely, a 'variable' (attribute) that has value or meaning for an 'instance', or object
  - '**class variable**', then, is often used as a synonym for 'static attribute'



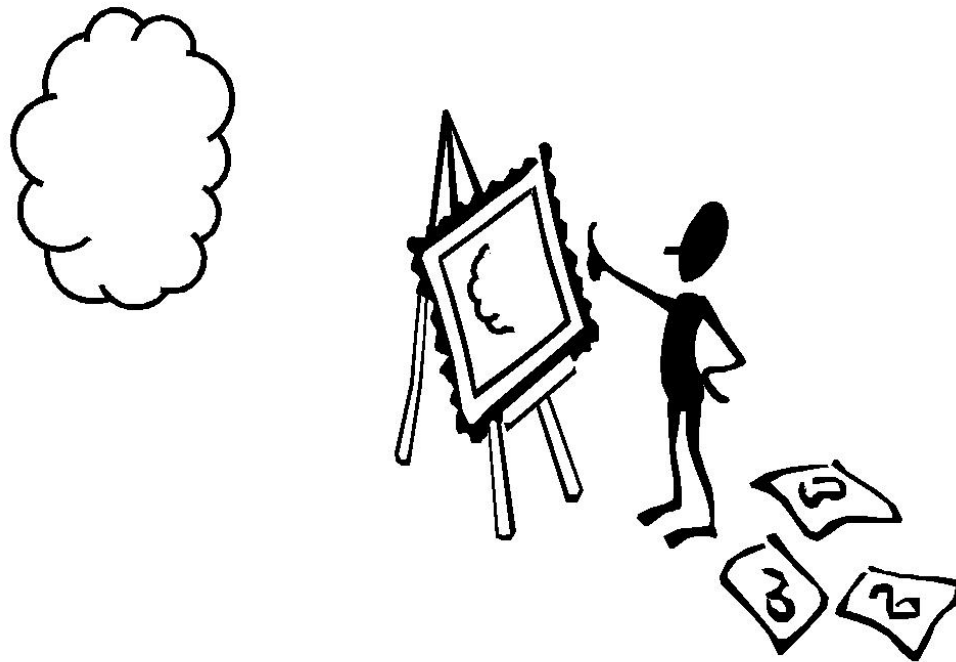
# Victory!



We've made it through all of the major object concepts that you need to understand in order to appreciate Java! (Part 1 of the book)  
(Part 3 refines this knowledge ...)

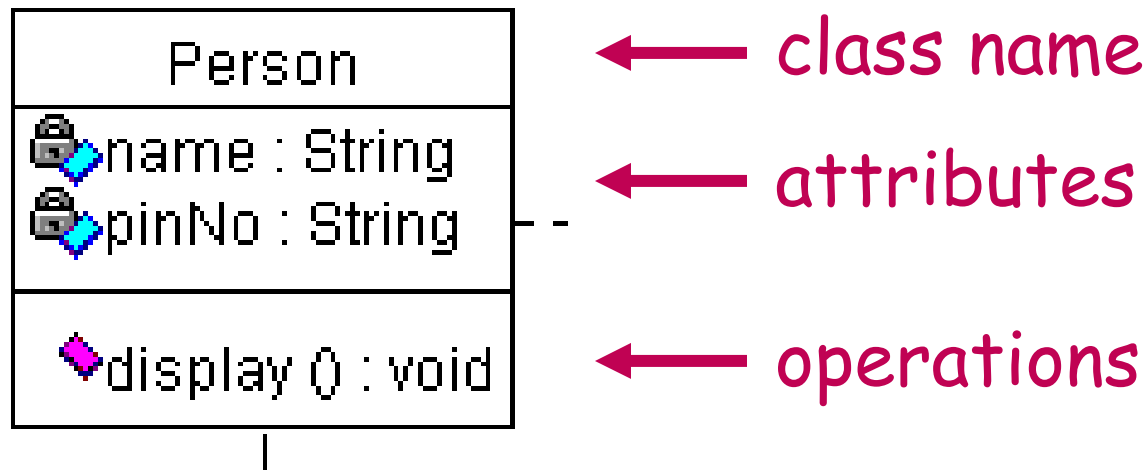
# A Quick Review of UML Notation

(Chapter 10)



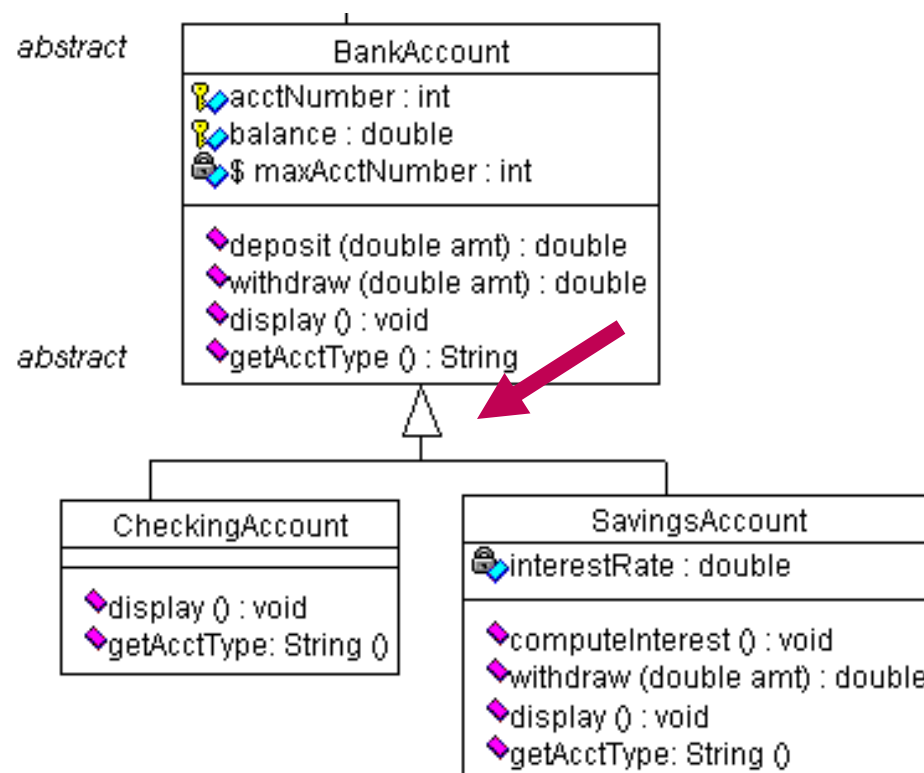
# UML Building Blocks

- Class rectangle is subdivided into three compartments



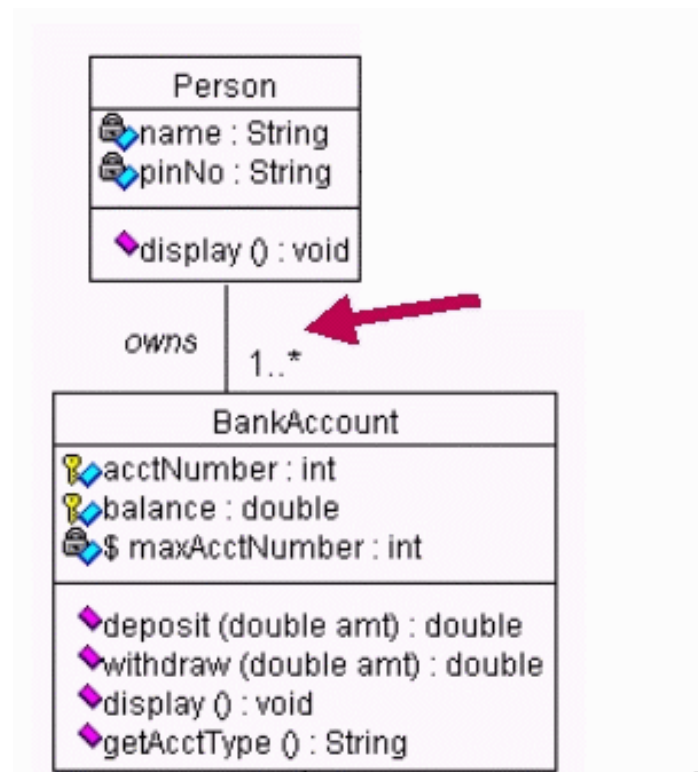
# UML Building Blocks, cont.

- Inheritance: triangle pointing at superclass



# UML Building Blocks, cont.

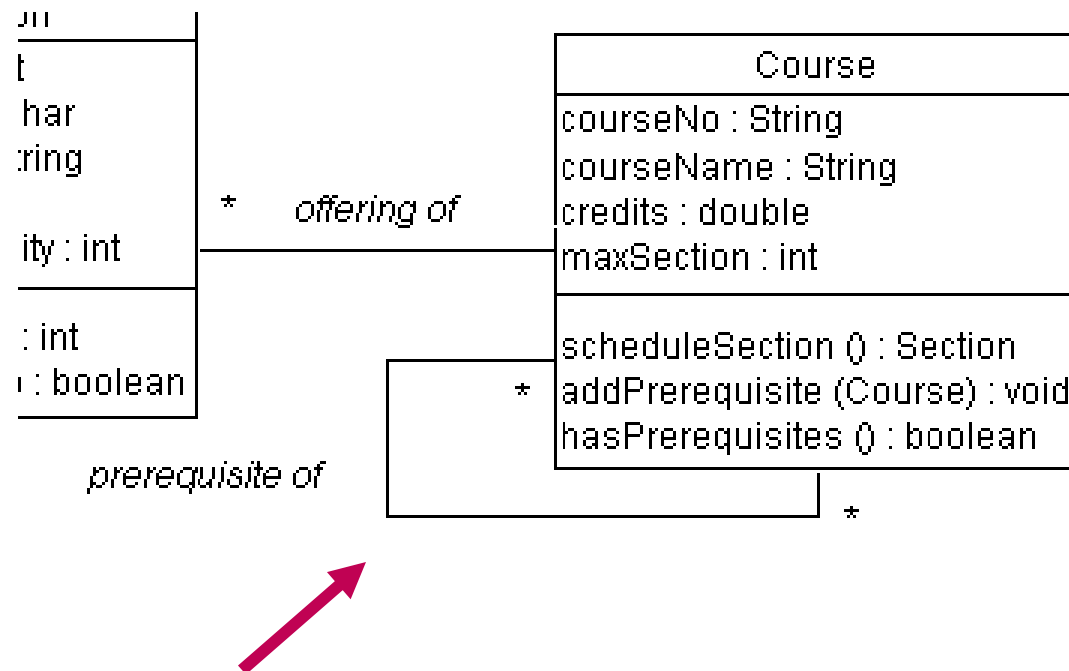
- Association: labeled line connecting classes



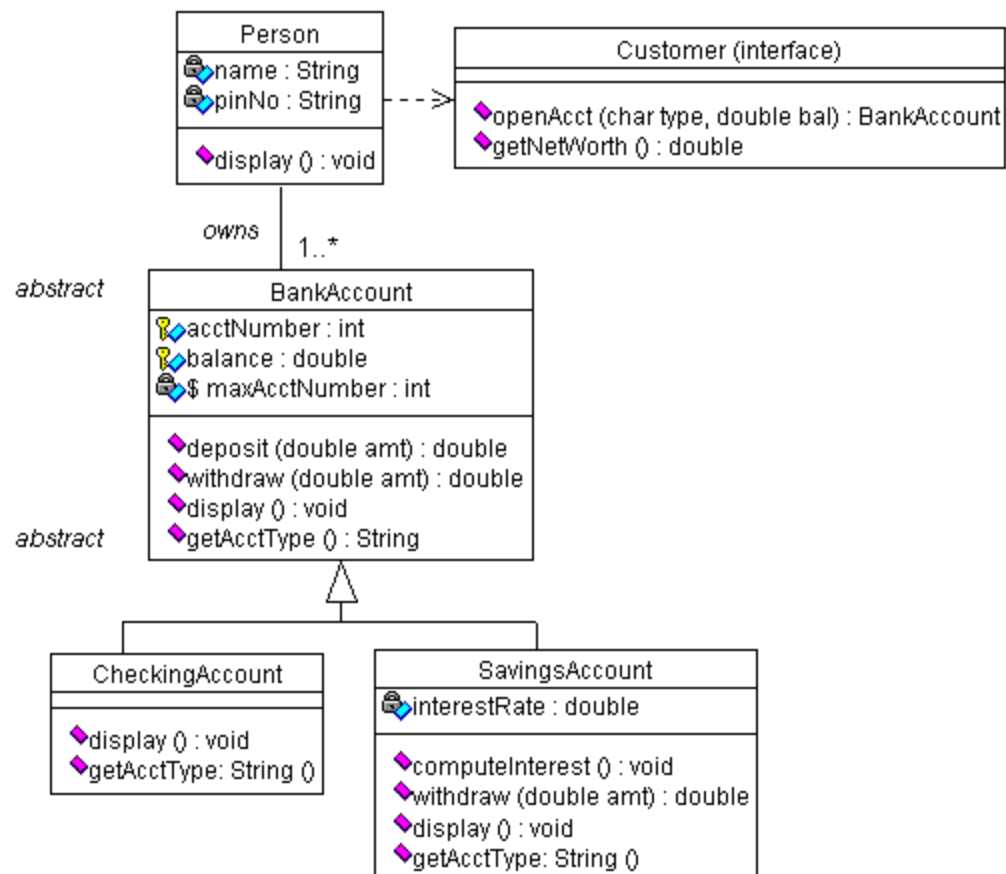
- Multiplicity labels: `0..*`, `1..*`, `*`, (plain = 1)

# UML Building Blocks, cont.

- Reflexive association “loops back”



# Our Banking Application



# Banking Application Model

- Of course, we'd normally start with such a UML model -- before we write any code -- to serve as a "blueprint"
- Part 2 of my book (chapters 8 - 12) discuss the basic principles of object modeling



# Rounding Out a Java Application

# Rounding Out Our App.

- So far, we've learned about the power of Java as an OOP to model real-world objects in software
  - Person, BankAccount, SavingsAccount, etc.
- We developed a command-line driven banking application, BankingApp.java
- BankingApp.java was somewhat simplistic
  - No graphical user interface (we ran the program from the DOS command line: `java BankingApp`)
  - All objects (data) were "hardwired" (versus reading from a database or file)  

```
Person p = new Person("Fred", 123456);
```
  - No way to persist objects' states once the application terminated (state = collective values of all attributes)

# Rounding Out Our App., cont.

- We're going to finish up our Java studies by talking about how to round out such a model to create an "industrial strength" Java application
- This requires:
  - A way to present this model to a user such that he/she can interact with/command objects (typically, via a graphical user interface)
  - A way to persist (permanently store) the state of our objects from one application session to the next

# Object Persistence

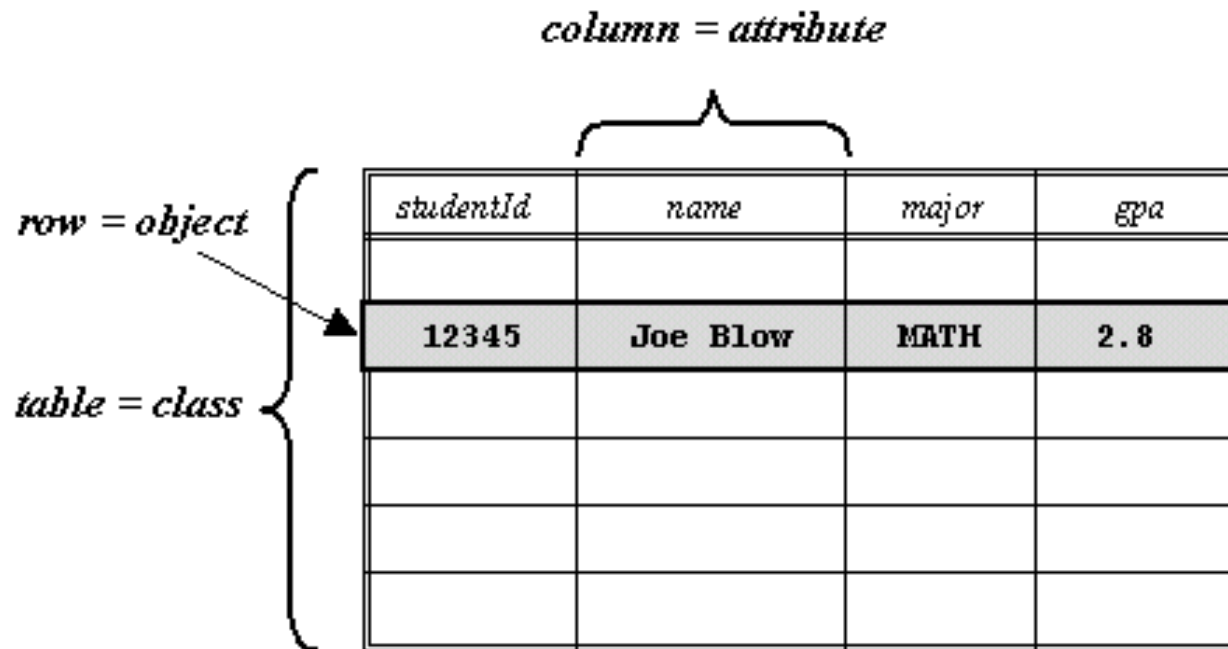
- When we instantiate objects at run time, these objects live within the memory allocated to the Java Virtual Machine (JVM)
- When the application terminates, all memory allocated to the JVM is reallocated to the operating system, and our "in memory" objects are lost
- If we wish for the objects' states (attribute values) to persist across application sessions, we must save them to some sort of persistent storage

# Ways of Persisting an Object's State in Java

- There are numerous ways to persist an object's state; some of the more common:
  - In files
    - In text files with a record-oriented structure -- aka "flat files"
    - As XML documents (also text)
    - As binary objects in files (serialization)
  - In databases
    - **As traditional records in a relational database (most common)**
    - As binary objects in relational databases (recent)
    - As binary objects in object databases

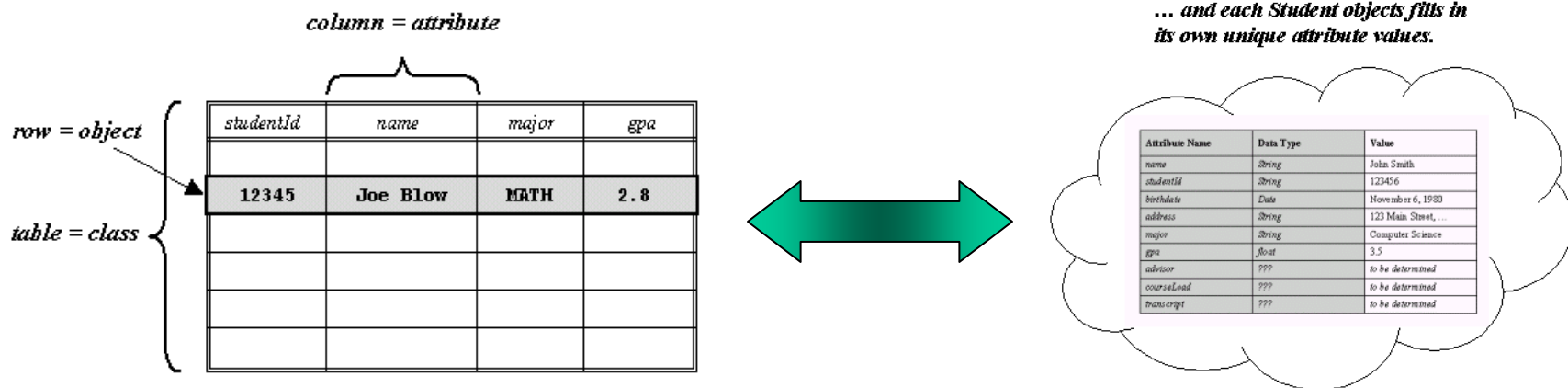
# Objects and Databases

- The concept of objects/classes naturally relate to the concepts of records/tables in a database

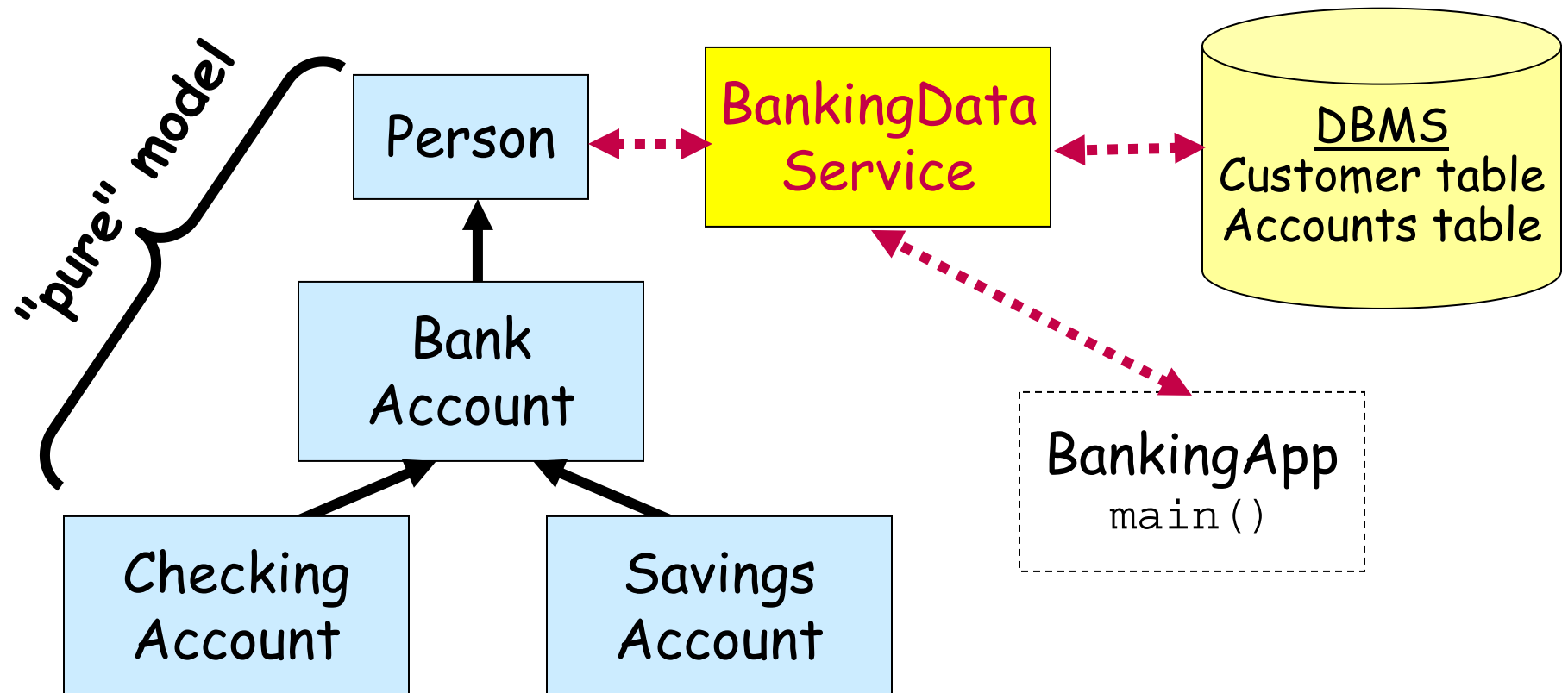


# Objects and Databases, cont.

- We persist/restore the state of an object with a relational database by creating special **data access layer classes** that provide methods for reading from/writing to a database



# Model - Data Layer Separation





# BankingDataService as a Utility Class

- We're going to engineer BankingDataService as a utility class
  - Similar to the Math class of Java
  - All methods will be static, so that we will be able to call them on the class as a whole:

`BankingDataService`.getPerson("123-45-6789")

- Similar to: `Math.sqrt(25.0);`

# BankingDataService.java

```
public class BankingDataService {  
    // This 1st method is used to call the Person constructor  
    // behind-the-scenes, populating a new Person object with  
    // data retrieved from a database based on the person's  
    // unique identifier.  
    public static Person getPerson(String ssn) {  
        Person p = null;  
  
        // Pseudocode.  
        use ssn as a primary key to look up this person's  
        record in the database's Customer table;  
  
        if (record found) {  
            retrieve the record;  
  
            // Instantiate a new Person object based on  
            // this record.  
            Person p = new Person(...);  
        }  
        return p;  
    }  
}
```

# BankingDataService, cont.

```
// This 2nd method is used to store the data for a given
// Person back to the database.
public static void persist(Person p) {
    // Pseudocode.
    use ssn as a primary key to look up this person's record
    in the database's Customer table;

    if (record found) {
        update the record based upon the in-memory
        attribute values of this object;
    }
    else {
        insert a new record in the table representing
        this object;
    }
}
} // end of class
```

# BankingApp.java

- Here's our main program, modified to take advantage of DBMS access:

```
// BankingApp.java.  
public class BankingApp {  
    public static void main(String[] args) {  
        try {  
            // Instantiate one Person object using our new service  
            // method.  
            Person p = BankingDataService.getPerson("123-45-6789");  
  
            // Use the display() method to check the results.  
            p.display();  
  
            // Create another account for this customer. (As it turns  
            // out, he already had two accounts -- no. 4 and no. 5.)  
            CheckingAccount c = new CheckingAccount(p, 6, 200.0);
```

# BankingApp.java

```
// Retrieve account no. 4, and make a deposit.
BankAccount b = p.getBankAccount(4);
if (b != null) b.deposit(1000.0);

// Persist the new state of this customer's bank accounts
// back out to his account file (nnn.dat).
BankingDataService.persist(p);
    }
}
}
```

# JDBC API

- The pseudocode in the preceding examples would be written as real Java code using the Java DataBase Connectivity (JDBC) API
- JDBC provides us with a **vendor independent** way of connecting to and querying/updating ODBC databases using Standard Query Language (SQL) syntax

```
String query = "SELECT * FROM Customer where ssn = " + ssn;  
ResultSet rs = stmt.executeQuery(query);  
// etc.
```

## JDBC API, cont.

- We can write an application to access a Sybase database today
- If we need to migrate the database to Oracle tomorrow, our application will work "as is"
- Caveat: avoid vendor specific "bells and whistles" that go above and beyond the Sun's core JDBC specification!

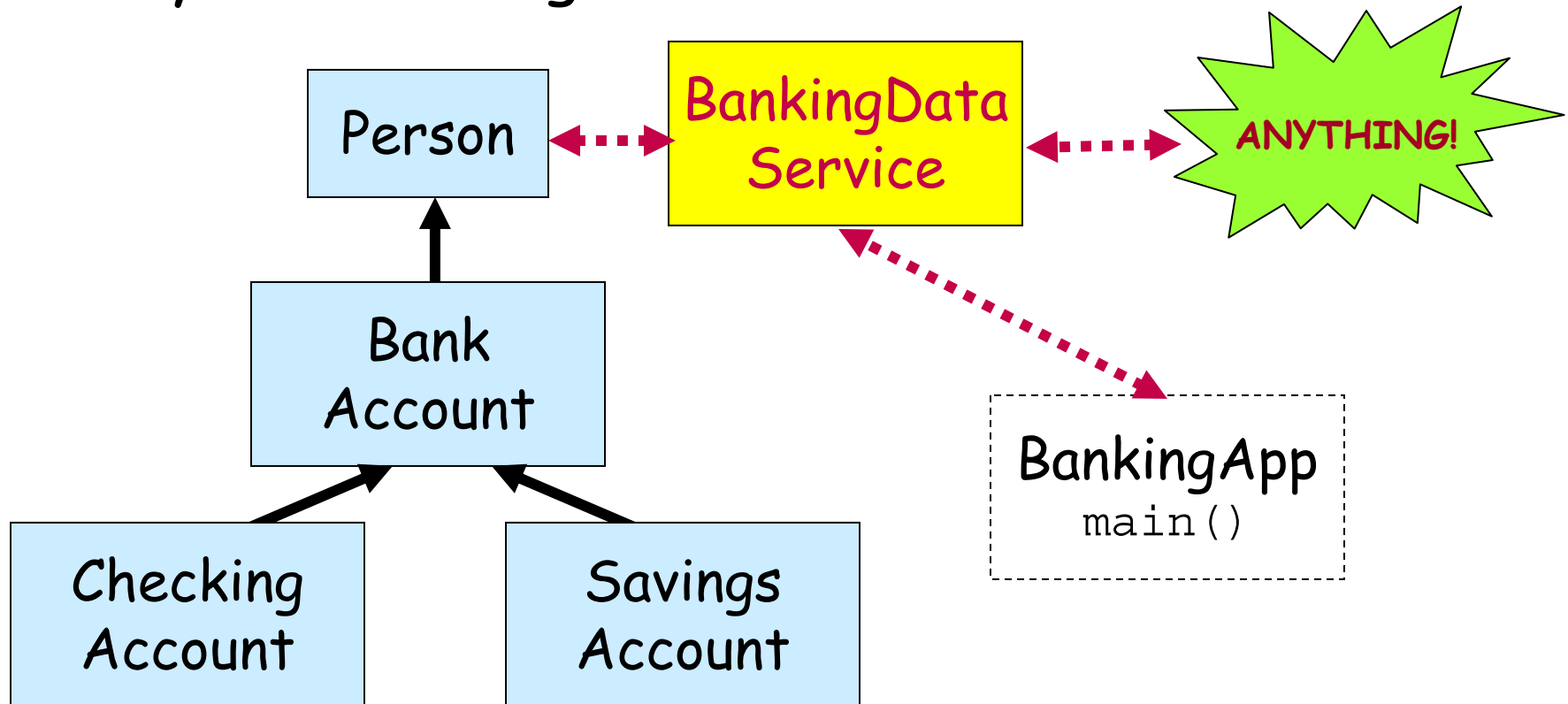
## Model - Data Layer Separation, cont.

- Better yet, we can totally replace the persistence mechanism, abandoning relational databases for an entirely different technology if we choose!



# Model - Data Layer Separation

Only the BankingDataService class is affected!



# Model - View Separation

- Let's now turn our attention to how we graphically present the model to our users
- A technique known as **separating the model from the view** is an important design approach when developing a graphically-oriented application
  - Relates to the **Model-View-Controller (MVC) paradigm**, which was popularized as a formal concept with the Smalltalk language
  - Also known as separating the presentation logic from the business logic

# Model - View Separation, cont.

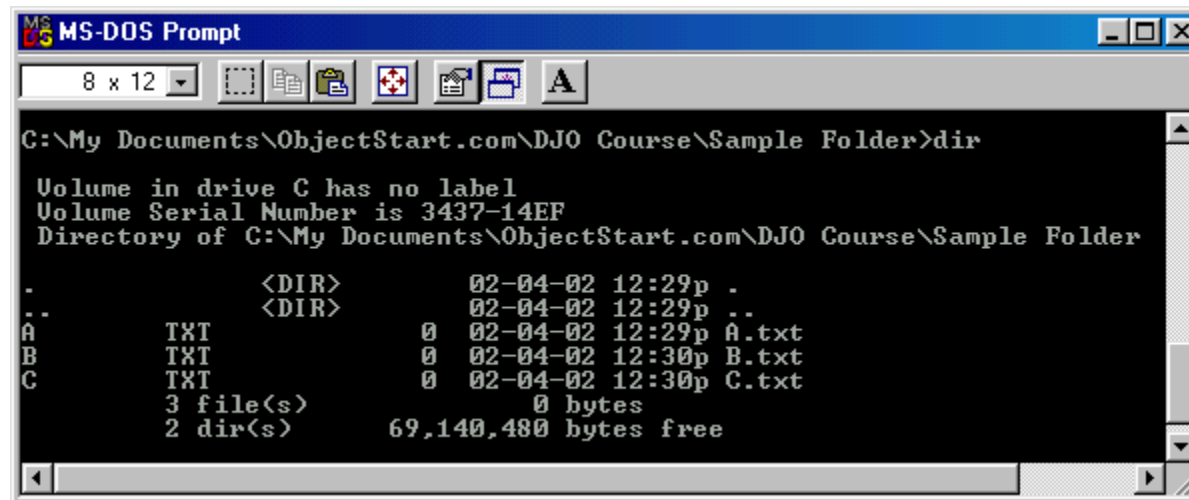
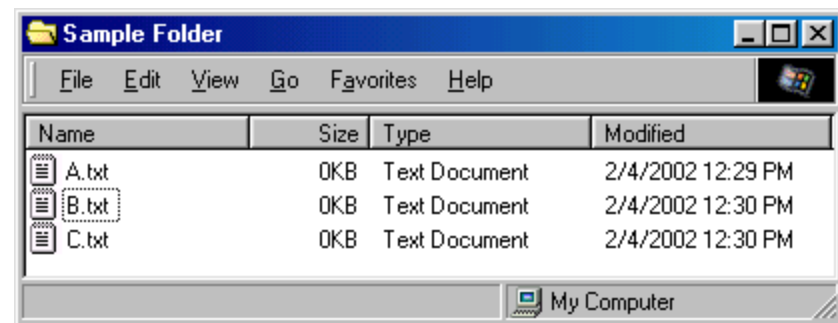
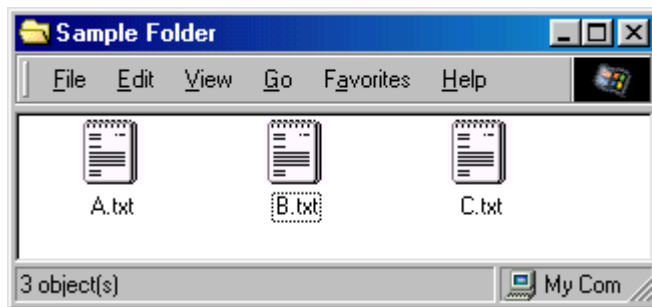
- The **model** embodies the **domain knowledge** of the application:
  - The objects/classes that represent the real-world objects that users are familiar with
    - Person, BankAccount, SavingsAccount, CheckingAccount
  - Their interrelationships and interdependencies

# Model - View Separation, cont.

- The **view** is the way in which we present this knowledge to the user - typically, although not exclusively, via a graphical user interface
- There can be many different views of the same model, in the same or different applications
- Example: the real-world notion/model that a Windows folder called "Sample Folder" contains three files/documents called "A.txt", "B.txt", and "C.txt" ...

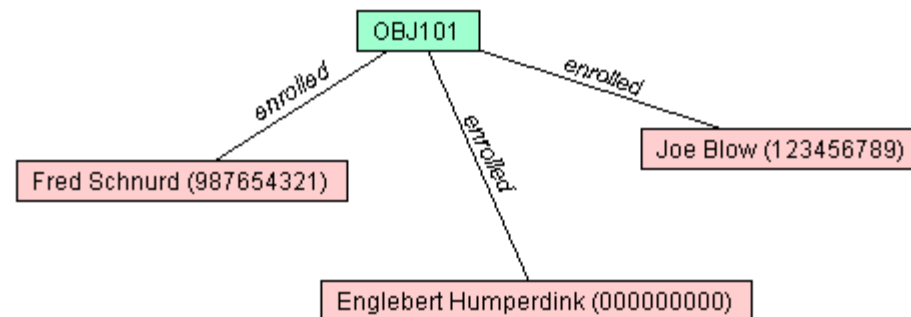
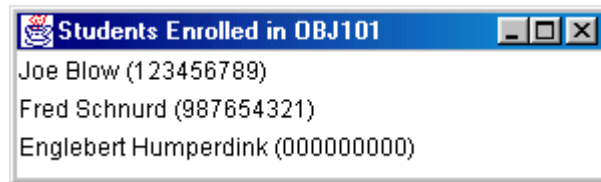
# Model - View Separation, cont.

... can be presented in Windows with many different views!



# Model - View Separation, cont.

- Or, in the case of the SRS:



# Technique

- By programming the model first, then tackling the view as a separate step, we ensure loose coupling between the model and the view
  - An important design technique for model reusability
- Too many beginning Java programmers approach this in reverse order!
  - The use of drag-and-drop IDEs allow users to build a an (object-oriented) view against a non-OO back end!

# Ways of Presenting an Application's View in Java

- Conventional desktop applications
- Web-deployed applets
- Thin client web GUIs, built utilizing J2EE component technology



# Desktop Applications

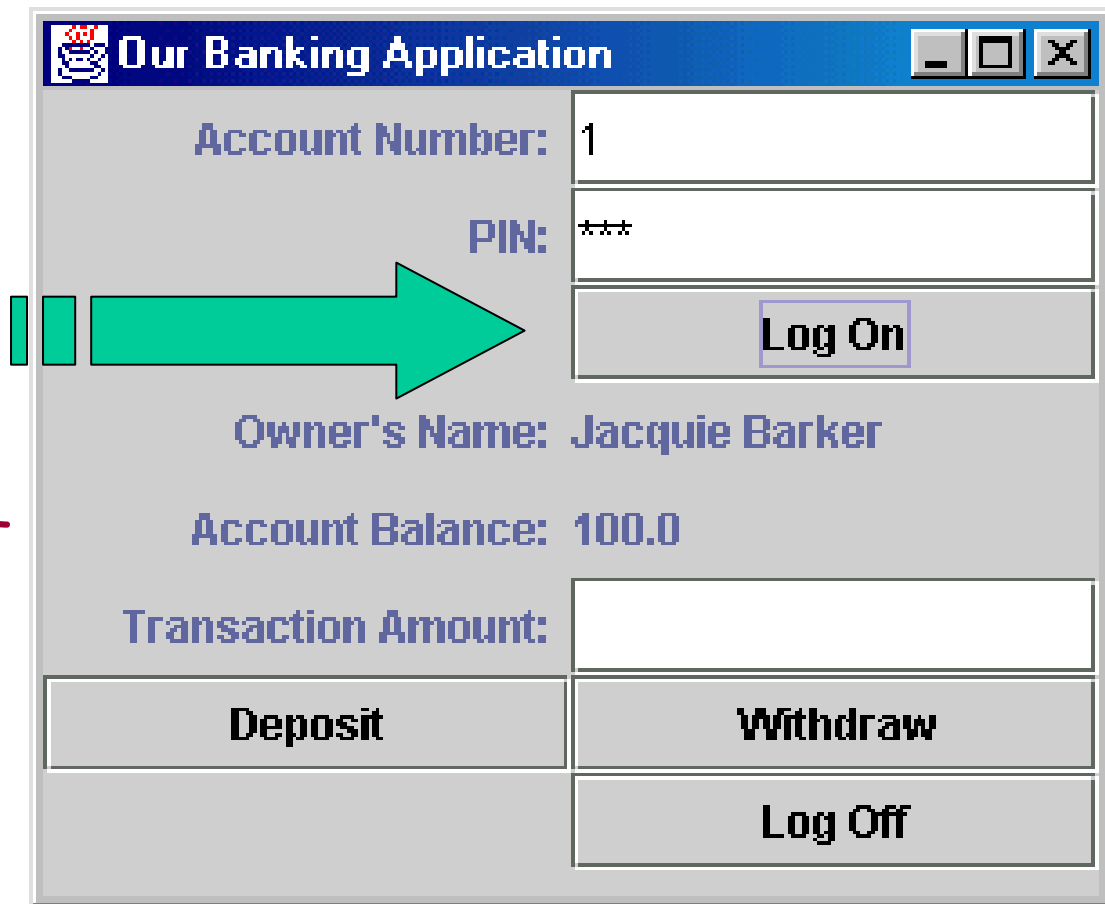
- The fundamental approach to desktop GUI programming in Java is to assemble graphical building blocks -- buttons, text fields, lists, menus, etc. -- known as **components**
  - Java provides two built in libraries/APIs -- the Abstract Windowing Toolkit (AWT) and Swing -- which jointly define all of the necessary components
  - We assemble components in specific ways to provide the 'look', or presentation, that we desire for an application
  - We then program their 'behind the scenes' logic to enable them to do useful things - this is known as **event handling**

## Components, cont.

- Users then interact with these components to "drive" the underlying model objects -- in our case, Person, BankAccount, CheckingAccount, and SavingsAccount objects -- to achieve a particular goal
  - Instantiate objects
  - Establish links among them
  - Invoke their methods
  - Change their state (attribute values)

# Sample Banking App GUI

Click here, and two objects -- a Person object representing the user and a BankAccount object representing the account of interest -- are instantiated



Our Banking Application	
Account Number:	1
PIN:	***
<b>Log On</b>	
Owner's Name: Jacquie Barker	
Account Balance: 100.0	
Transaction Amount:	
<b>Deposit</b>	<b>Withdraw</b>
<b>Log Off</b>	

# The Downside of Desktop App. Deployment

- Desktop applications are disseminated in executable form (byte code, in the case of Java) to the individual workstations on which they are to run
- There are numerous "headaches" associated with deploying application code in this fashion

# Desktop Apps., cont.

- Issue #1: there is potentially a lot of software infrastructure that must exist on the client machine in order for the application to run
  - JVM; DLLs
  - "Fat client"
  - Will the client machine have enough "horsepower"?
  - Debugging the installation/configuration of all of this software can be a nightmare

# Desktop Apps., cont.

- Issue #2: there are risks as to whether or not all users will get subsequent updates after the application is deployed
  - Must keep track of everyone who is using a given application
  - Must alert them to new releases
  - Have to rely on the user's initiative to install or download the new version
  - May lose track of some users
  - A configuration management/support nightmare can ensue!

# Web Deployment

- Web deployment is an alternative to physically distributing executable code
  - Whenever a user wishes to use an application, he/she accesses the URL of the site hosting the application via his/her web browser
  - The application is delivered "just in time" to that user's browser for his or her use at that moment -- always guaranteed to be the newest release
  - Analogy: distributing/watching a videotape of daily news vs. watching "live" CNN broadcast on TV

# Web Deployment, cont.

- No specialized infrastructure is, in theory, required to distribute applications
  - Use the existing Internet/intranet infrastructure and an already present web browser
  - Particularly important with public domain applications
- CM problems resolved
  - Code is maintained centrally
  - Only those who use the application receive the latest version just in time ("demand pull")
  - Conversely, people who don't receive it by definition didn't need/use it!



# Web Deployment, cont.

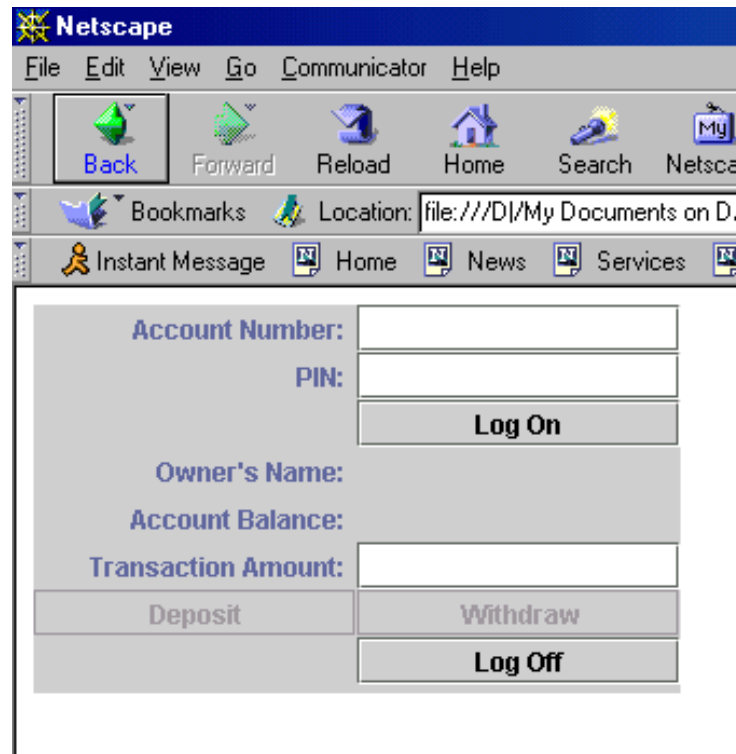
- Broader potential audience for an application
  - People can accidentally discover an application if it isn't access restricted
- It's no surprise that large IT organizations have migrated toward web deployment of applications whenever feasible

# Applets

- An applet is a downloadable Java program referenced by a special tag within an HTML page
  - When you visit such a page, the code of the applet is automatically physically downloaded to your local (client) computer
  - Run under control of your browser's (built-in) JVM (requires a plug-in)
  - Applets have GUI interfaces -- built from the same AWT/Swing components (for the most part) as desktop applications

# Running an Applet

- "Surf" to its HTML home page, and the applet appears (IF the appropriate plug-in is present)



# Applets are "Thick"

- Despite the fact that they are web-deployed, applets are considered to be "thick"/ heavyweight clients, because of the amount of code and supporting data that has to be resident on/downloaded to the client machine
  - Plug-in(s)
  - Applet's byte code

# Problems with Applets

- Can be slow to download
  - Creates undue load on underlying intranet network
- Can be slow to start-up ("Loading Java ...")
- Can be slow to run, if client machine is underpowered
- Java plug-in may be missing from browser
  - Slow to download
  - Users may not want to bother, or may be intimidated
- Incorrect version of the plug-in may be installed
  - Unpredictable results!
- Bottom Line: too much reliance on client

# One Advantage

- Applets have one advantage over desktop apps built with Swing: “just in time” bytecode retrieval

# Are Applets "Dead"?

- No ...
  - Sophisticated GUIs may still need the interactive 2D/3D graphics capabilities of Swing, and hence applets are the only option for web deployment
- ... but ...
  - The vast majority of applications don't need heavy duty interactive graphics
- Think of the websites you've visited lately
  - Most e-commerce websites are heavily forms oriented, with "static" graphics
  - For these types of applications, thin HTML clients are preferred

# Thin Clients

- Definition: the client component of a client-server architecture is trivially simple
  - Involved primarily with displaying information (presenting the "view")
  - Virtually no business logic, no heavy duty processing occurs on the client machine
  - The server carries the lion's share of the processing load
- Classic thin client approach: a web browser to display HTML pages



# HTTP Protocol

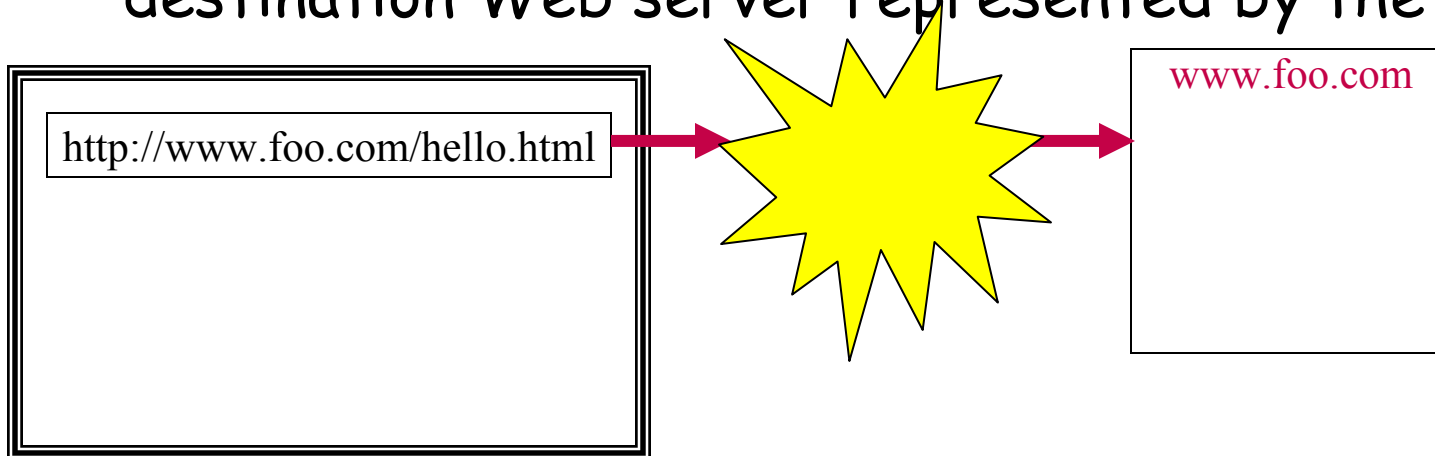
- HTML is transferred from a Web server to a browser via the HyperText Transfer Protocol (HTTP)
  - User types a Uniform Resource Locator (URL) into his/her browser



<http://www.foo.com/hello.html>

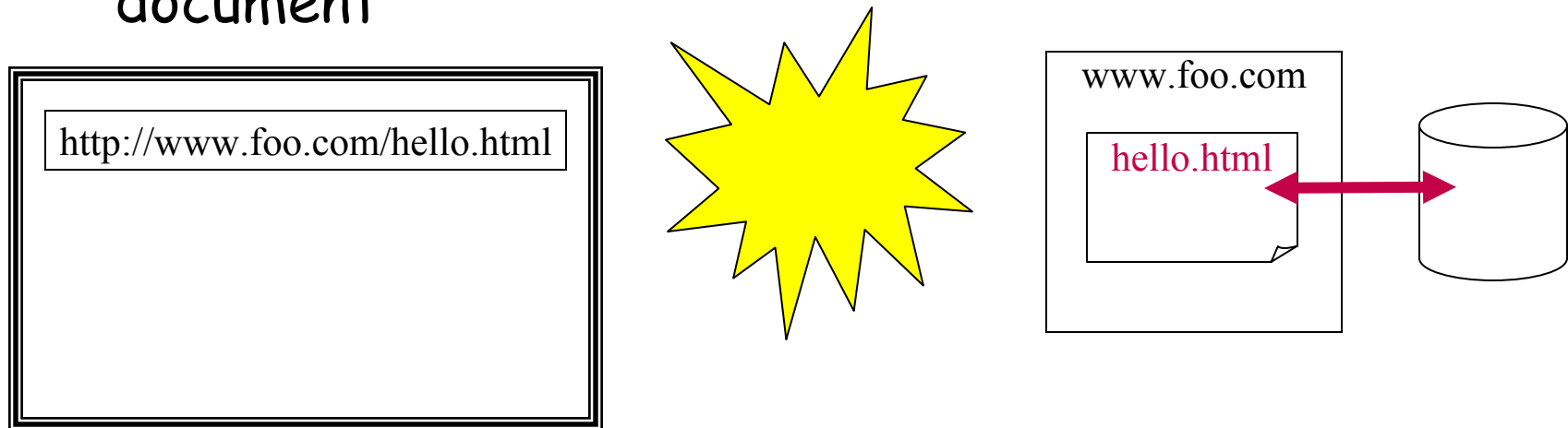
# HTTP Protocol, cont.

- HTML is transferred from a Web server to a browser via the HyperText Transfer Protocol (HTTP)
  - An HTTP **request** is sent via the Internet to the destination Web server represented by the URL



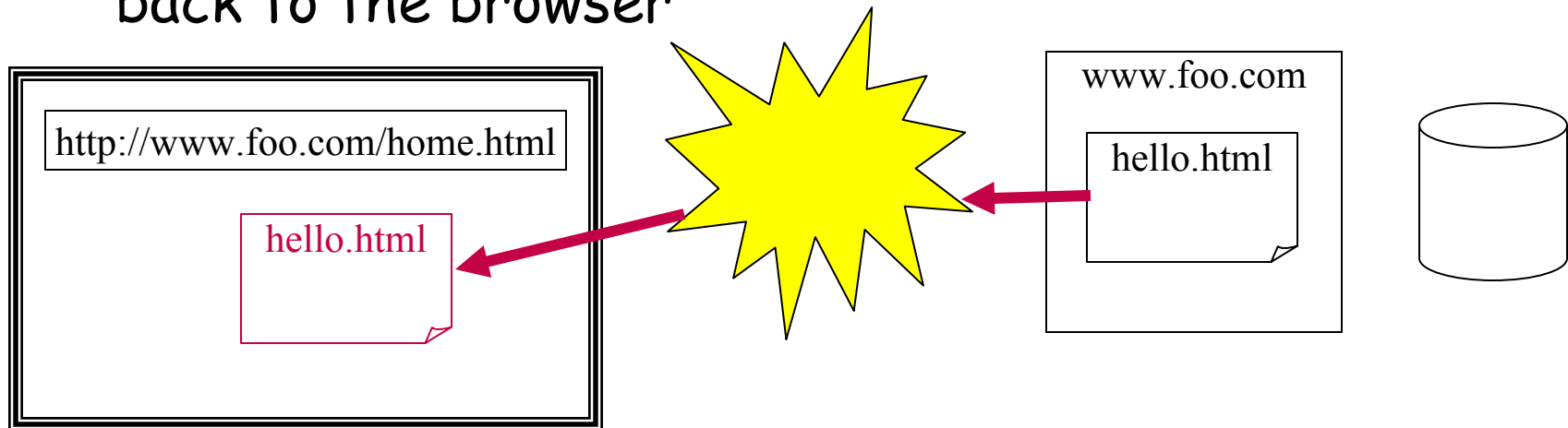
# HTTP Protocol, cont.

- HTML is transferred from a Web server to a browser via the HyperText Transfer Protocol (HTTP)
  - The Web server retrieves the requested HTML document



# HTTP Protocol, cont.

- HTML is transferred from a Web server to a browser via the HyperText Transfer Protocol (HTTP)
  - A **response**, containing the HTML document, is sent back to the browser



# HTTP Protocol, cont.

- Browsers understand how to interpret HTML tags so as to display a Web page's content to us in all of its glory:



# What is J2EE?

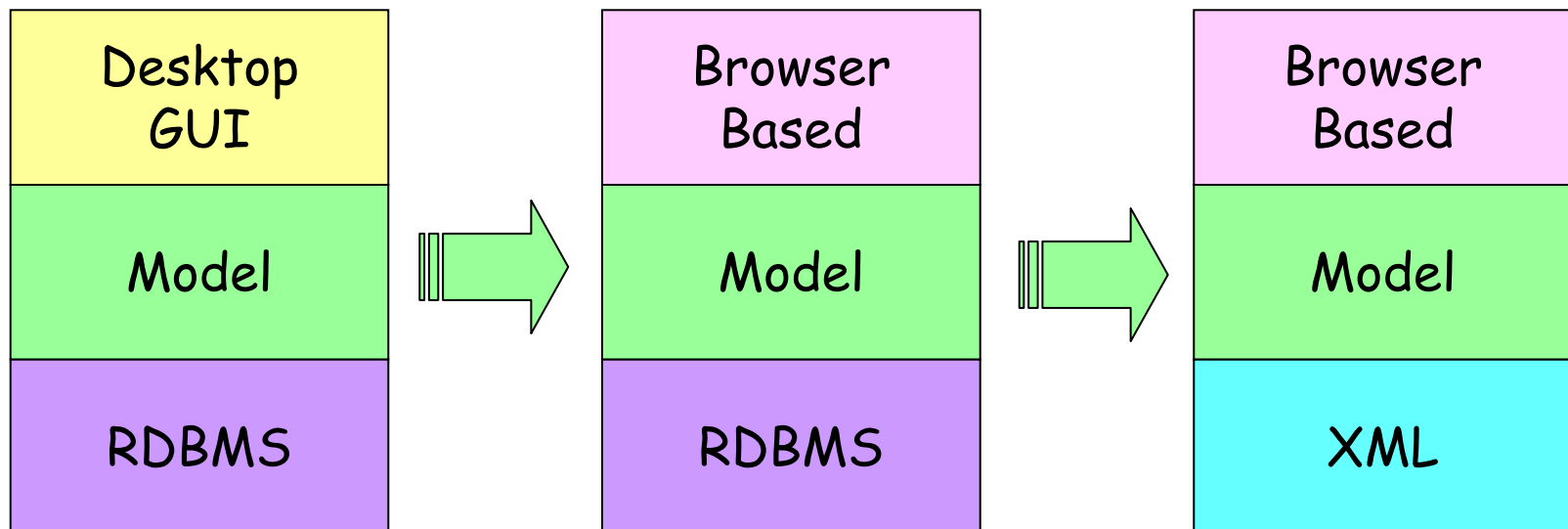
- Java 2 Enterprise Edition (J2EE) is a collection of eleven (11) separate Java component technologies/APIs that can be used separately or in combination to facilitate enterprise-level distributed applications
  - **Servlets, JDBC, and JavaServer Pages** are three of the most widely used J2EE technologies according to the Gartner Group
  - Others: **Enterprise JavaBeans (EJBs)**, Java Naming and Directory Interface (JNDI), Java IDL, Java Message Service (JMS), Java Transaction API (JTA), Java Transaction Service (JTS), JavaMail, RMI-IIOP

## J2EE, cont.

- J2EE takes advantage of the HTTP protocol as the "backbone" for delivering the **view** of an application via a user's browser
- The other aspects of the application - namely,
  - Business logic/model layer
  - Data access logic/persistence layerreside on a (web/application) server

# Importance of Model-View and Model-Data Layer Separation

- Can modernize presentation and persistence technology while preserving business rules





# Relevance of This Course

- Programming the model first, and then writing a command-line driven driver program to exercise the model, is a useful approach for debugging our model
- By tackling the data layer and view as separate steps, we ensure loose coupling between the model, the view, and the data layer
- You've learned enough in this course to be able to develop "bean" classes in support of a J2EE application team

# Follow On Course

- In my follow-on course, Deploying Java Objects, we cover the following topics related to "rounding out" an industrial strength application
  - Providing a graphical user front end
    - Using the SWING API (for standalone applications and Web-deployed applets)
    - Using HTML, JavaServer Pages (JSPs), and the Servlet API for thin-client Web-deployed J2EE applications
  - Providing object persistence
    - Using file I/O
    - Relational databases and the JDBC API

# Follow On Course, cont.

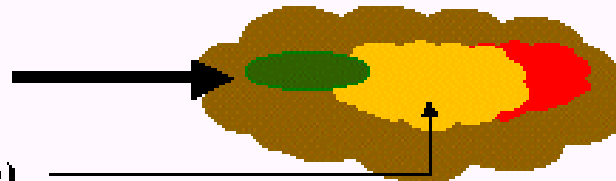
## An “Object Burger”

The front-end  
GUI



The “meat”:  
the object model

*(condiments = solution space code)*



The back-end  
database



# Suggestions for Further Study

(Chapter 17)



# "Homework Assignments"

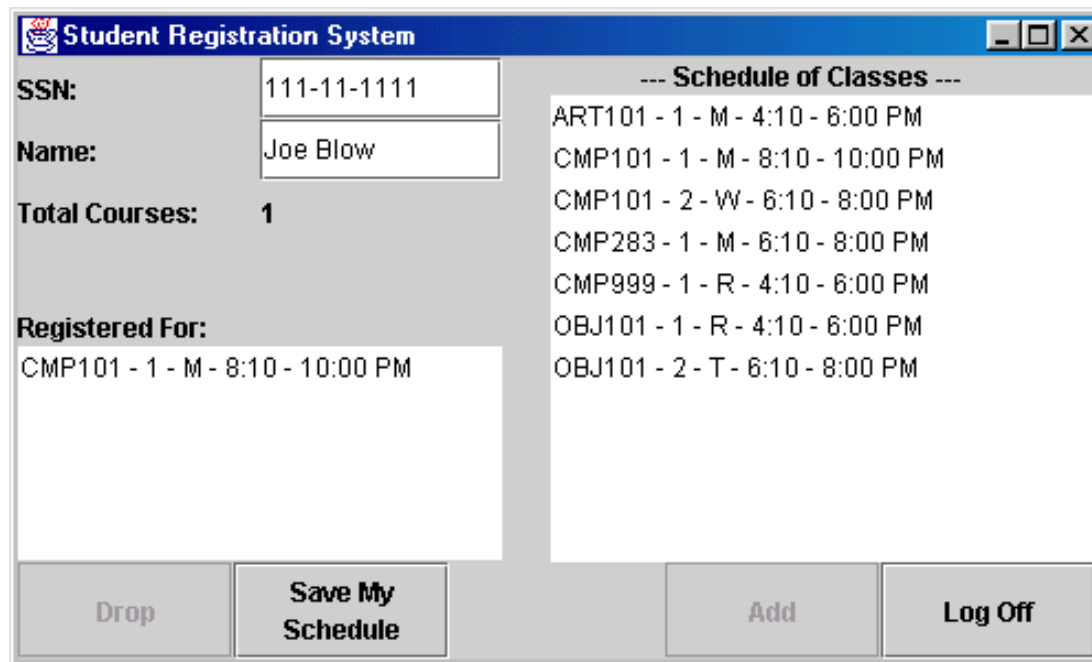
- Read the rest of my book!
  - Skim quickly through chapters 1-7
  - Read Part 2 (chap. 8 - 12) about object modeling
  - Read Part 3 (chap. 13 - 17) for more Java details
  - Visit my website -- [www.objectstart.com](http://www.objectstart.com) -- for errata
- Get your hands dirty!
  - See Appendix C for instructions on how to download the J2 SDK - it's free! :oD
  - Download the SRS code from [www.wrox.com](http://www.wrox.com)
  - Do end of chapter exercises
  - Pick a *SMALL* sample problem to automate

# "Then What?"

- Learn more details about the "core" Java language
  - Beginning Java 2 -- Ivor Horton
  - Thinking in Java -- Bruce Eckel
  - Java: An Introduction to Computer Science and Programming -- Walter Savitch
  - Sun/Prentice Hall "Core Java" Series of books
- Learn more about object modeling and UML
  - The Addison Wesley series of Unified Modeling Language reference books by Rumbaugh, Jacobsen, and Booch
- Learn about design patterns

# "Then What?", cont.

- Learn about JDBC
- Learn more about Java desktop GUIs

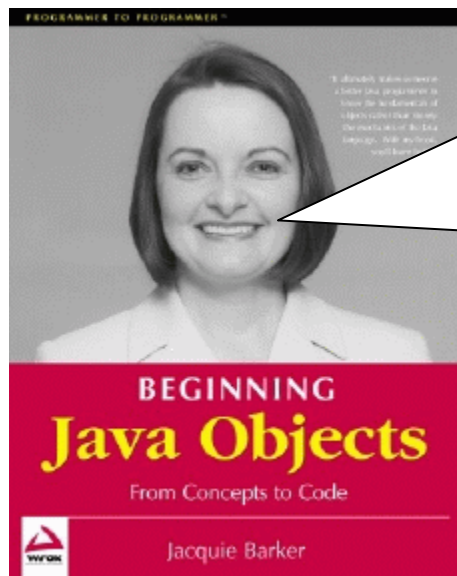


- Sun/Prentice Hall "Graphic Java" Series of books

# "Then What?", cont.

- Learn about Java and XML
- Learn about web deployment and J2EE
  - (Applets)
  - Servlets
  - Java Server Pages (JSPs)
  - (Optionally) Application servers/Enterprise Java Beans (EJBs)
- Make sure to visit [java.sun.com](http://java.sun.com)
- Find a mentor to guide you!





Thanks for your  
interest  
and support!  
Please visit me via my  
website at  
[objectstart.com](http://objectstart.com).

*object*start.com

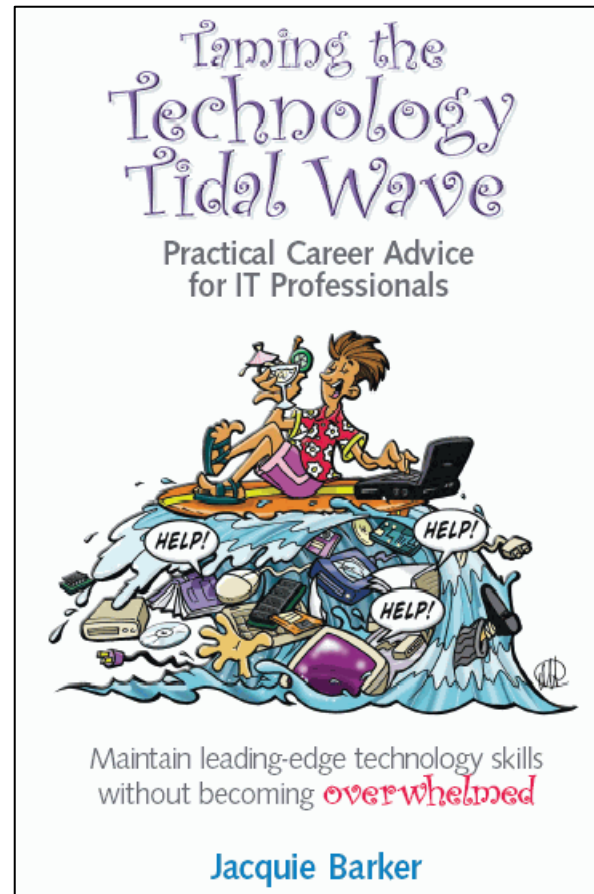
Jacquie Barker

Sun Microsystems Certified Java 2 Programmer  
Object Technology Consultant/Instructor  
Author, Beginning Java Objects

9214 Bayard Place  
Fairfax, VA 22032

Voice: (703) 323-3972  
FAX: (703) 323-3962  
[jjbarker@objectstart.com](mailto:jjbarker@objectstart.com)

# Based On:



(ISBN 0974479888)

11/11/2009

Copyright © 2005 Jacquie Barker

635

## Attention all IT professionals (a.k.a. "techies"):

- Are you utterly **overwhelmed** by how fast technology changes?
- Are you **exhausted** from running as fast as you can to learn the latest "technology du jour" ... only to find out that by the time you've mastered it, it's already **obsolete**?
- Do you worry that you've fallen so far behind on the technology curve that you'll **never catch up???**



# The Power of Paradigms

- A *paradigm* is a frame of reference, a fundamental way of thinking of things
- A *paradigm shift* is a major change in that world view
- In any long-standing field of technology, there are very few true innovations/paradigm shifts
  - Come along rather infrequently, but have a major, permanent impact on our world view
- The tools and methodologies that are used to *implement* these paradigms, on the other hand, change continually

# The Power of Paradigms, cont.

- In over 25 years as a software engineer, I've seen only a handful of new paradigms emerge
  - Modular ("goto-less") programming (early 70s)
  - Relational database management systems (early 80s)
  - Personal computers and client-server computing (mid 80s)
  - Graphical user interfaces (late 80s)
  - Object-oriented programming (late 70s, but not really commercially popular until the early 90s)
  - The World Wide Web (early 90s), and the move toward thin client computing (mid to late 90s)
  - The use of XML to represent/exchange data (mid 90s)
- I've seen *hundreds* of tools, languages, and vendor products come and go

# The Power of Paradigms, cont.



- Consider the speed with which the second hand on a clock face moves as compared to the hour hand:

- Blink, and we can miss a click or two's worth of movement of the second hand;

implementing technologies

- *Stare*, and we can't even see the hour hand moving, and yet we know that it does advance steadily

paradigm shifts

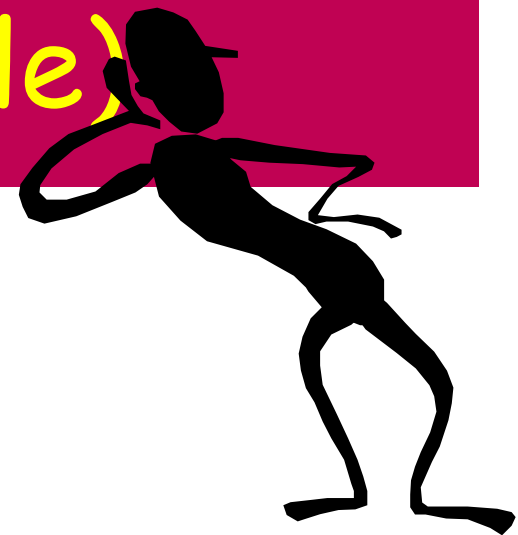
# The Power of Paradigms, cont.

- If we try to chase every new implementing technology, we'll invariably burn out
  - Never have enough time to truly get good at something before 'click!' - the second hand advances
- If we instead view our mission as **mastering paradigms**, our knowledge will serve us well for years to come
  - OO with OMT and C++ in early 90s



Now that you are hopefully sold on the power of objects ...

Why Java?  
(optional module)



# Why Java?

- Java's elegance as an OO language
- Platform independent, aka architecture neutral
- Vendor neutral
- Open standard

# Java is Vendor Neutral

- Java has seamless, built in language support for all aspects of application development
  - Database connectivity
  - GUI rendering
  - File/network access
  - etc. - no third party libraries are required
- If one stays within the class library framework that is provided by Sun Microsystems, applications remain vendor neutral

(cont.)

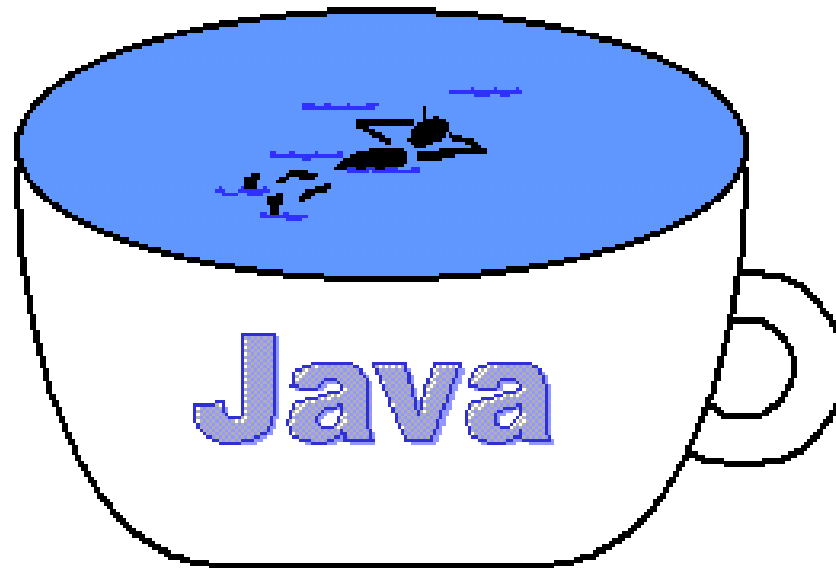
## Vendor Neutral, cont.

- We aren't "held hostage" by a particular vendor
  - Can switch from Sybase to Oracle
  - Can switch from BEA's WebLogic to IBM's WebSphere
  - Can switch from JBuilder to Forte IDEs
- Caution: beware of the use of vendor specific "bells and whistles" add-ons!

# Open Source

- Open source, and hence, low cost of entry
  - Virtually everything that you need to build "industrial strength" Java applications is freely downloadable from Sun's website and/or other community sites
    - Java software developers kits
    - Example applications
    - Reusable classes
    - Source code to everything!
    - Documentation
    - Tutorial information
    - Reference implementation servers

# Optional Module: Java in a Bit More Depth (Chapter 13)



# javadoc Utility

- As mentioned earlier, there is a special form of Java comment that can be used as the basis for automatically generated HTML documentation
  - Form: `/** slash double asterisk to start */`
- We use the javadoc command line utility, which comes with the Java SDK, to generate the HTML
- Let's look at an example ...

# javadoc Utility, cont.

```
// Person.java
```

```
/**
```

```
A person is a human being ...
```

```
*/
```

```
public class Person {
```

```
    // Attributes.
```

```
    /**
```

```
A person's legal name.  name is a public attribute.  Only  
public or protected attributes show up in automatically  
generated Java documentation.
```

```
    */
```

```
    public String name;
```



# javadoc Utility, cont.

```
/**
A person's age, rounded to the nearest year.
*/
private int age;

// Constructor.
/**
This constructor initializes attributes name and age.
@param n the Person's name, in first name - last name order.
@param a the Person's age, rounded to the nearest year.
*/
public Person(String n, int a) {
    name = n;
    age = a;
}

/**
This method is used to determine a person's age in dog years.
*/
public int dogYears() {
    return age/7;
}
}
```

11/11/2009

# javadoc Utility, cont.

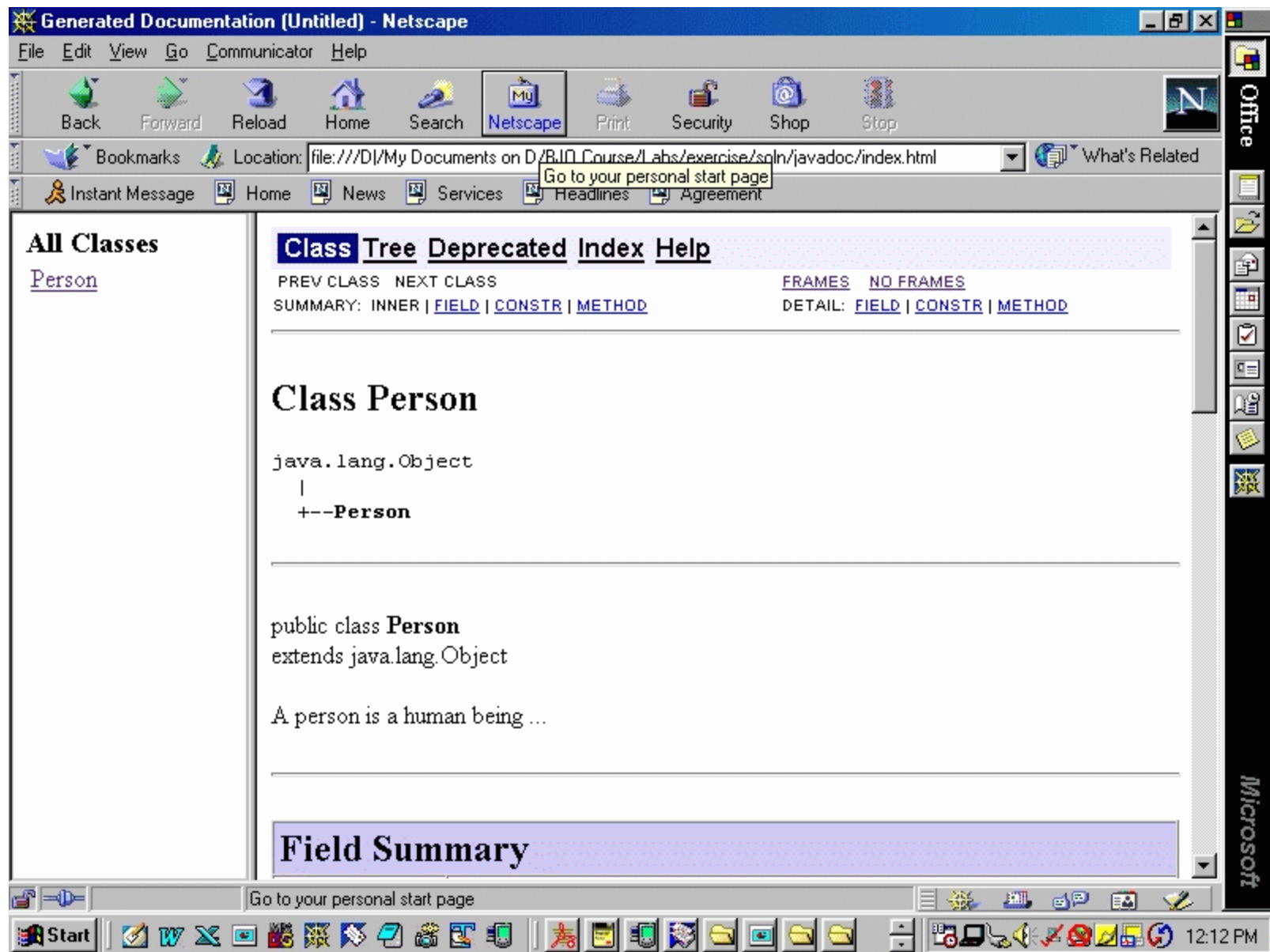
- To generate HTML documentation for this class, use the DOS/Unix command line command:

```
javadoc Person.java
```

or

```
javadoc *.java
```

- The resultant HTML is shown on the next several slides



Generated Documentation [Untitled] - Netscape

File Edit View Go Communicator Help

Back Forward Reload Home Search Netscape Print Security Shop Stop

Bookmarks Instant Message

Note: private features don't show up!

All Classes

[Person](#)

### Field Summary

java.lang.String	<a href="#">name</a>	A person's legal name.
------------------	----------------------	------------------------

### Constructor Summary

[Person](#)(java.lang.String n, int a)  
This constructor initializes attributes name and age.

### Method Summary

int	<a href="#">dogYears</a> ()	This method is used to determine a person's age in dog years.
-----	-----------------------------	---

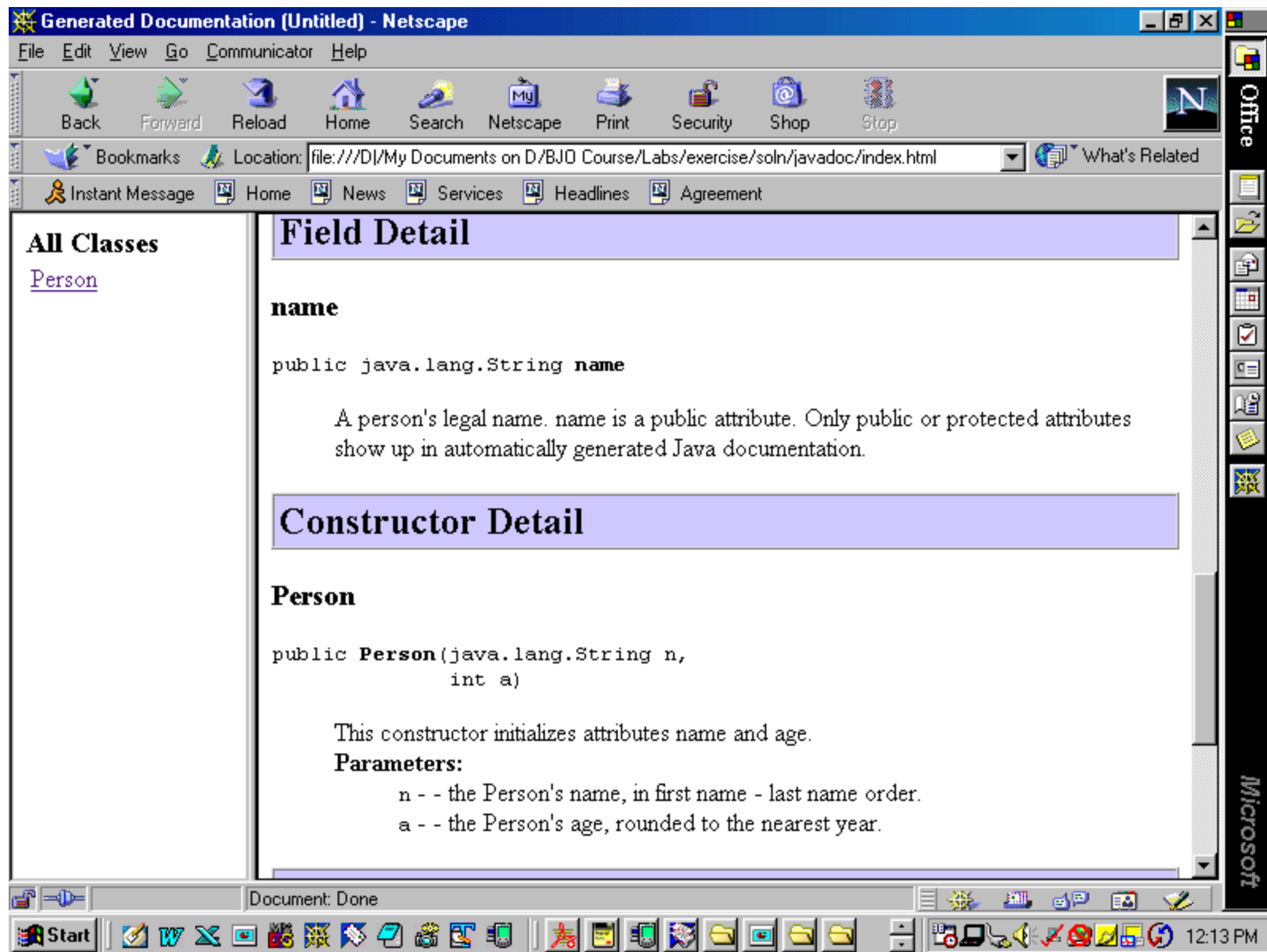
#### Methods inherited from class java.lang.Object

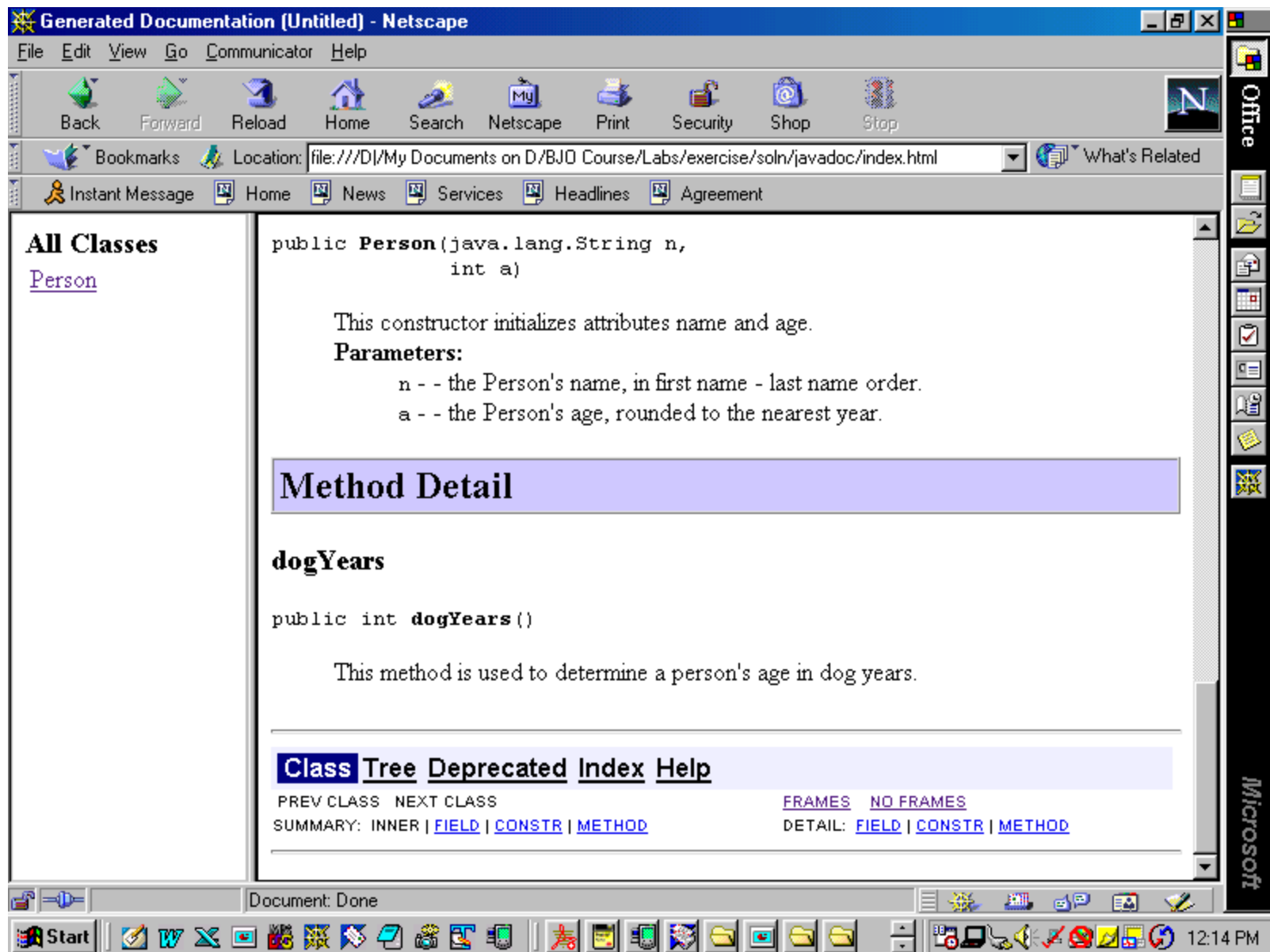
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Document: Done

Start

12:13 PM





## javadoc Utility, cont.

- See <http://java.sun.com/j2se/javadoc/> for all of the details on what can be done with this utility
- It is a powerful feature of Java -- make sure to take advantage of it!

# Strings as Objects

- We introduced the 'built-in' Java data types byte, short, int, long, float, double, char, boolean, and String earlier
- What we *didn't* make clear at the time is that Strings are *objects*, while all of the other types that we mentioned can be thought of as **primitive** (i.e. non-object) data types
- The structural and behavioral characteristics of String objects are, naturally, defined by the built-in String class



# Strings as Objects, cont.

- A sample String declaration and instantiation follow:

```
// Declare a reference variable to a FUTURE
// String - the String object itself
// doesn't exist yet.
String s;

// Now, we'll instantiate an actual String
// object.
s = new String("I am a string!");

// Shortcut:
s = "Another string";
// (the latter pulls from a String literal pool ...
// effectively "recycles" String objects)
```

# Strings as Objects, cont.

- As we saw earlier, the plus sign (+) operator concatenates Strings:

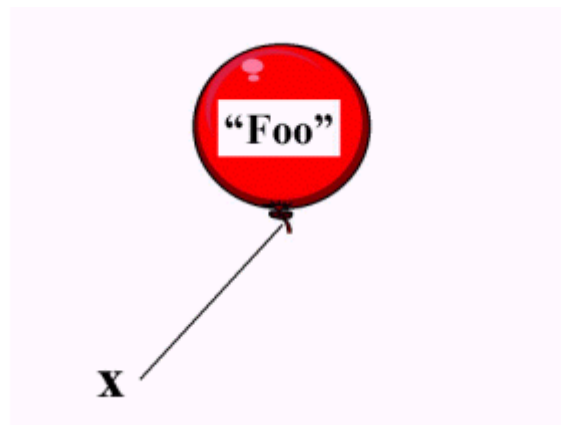
```
String x = "foo";  
String y = "bar";  
String z = x + y + "!"; // z now equals "foobar!"
```

- But now that we appreciate the object nature of Strings, we can also take advantage of the numerous methods that are available for manipulating Strings

# Immutability of Strings

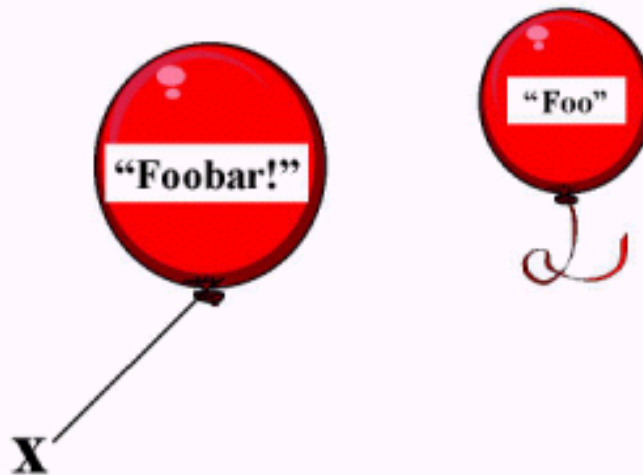
- Strings are **immutable**: that is, when you seem to be modifying the value of a String object, you are actually creating a new String object with the desired content:

```
String x = new String("Foo");  
// This results in a String object with the value  
// "Foo" being created somewhere in memory.
```



# Immutability of Strings, cont.

```
x = x + "bar!";  
// This results in a SECOND String object with the  
// value "Foobar!" getting created somewhere else in  
// memory; the original "Foo" String is still out  
// there, but like a balloon whose String we've  
// released, it is no longer directly accessible to  
// us by reference.
```



# StringBuffer

- The Garbage Collector will of course clean up all such discarded Strings

```
// Inefficient!  
String s = null;  
while (file not empty) {  
    s = s + next record from file;  
}
```

- If the file in question contained 50,000 records, however, then we'd have created 49,999 temporary String objects!
- If you are going to be doing heavy String manipulation, using a StringBuffer is more efficient

# StringBuffer, cont.

- StringBuffer examples:

```
// Efficient!  
StringBuffer sb = new StringBuffer();  
while (file not empty) {  
    sb.append(next record from file);  
}  
String s = sb.toString();
```

---

```
StringBuffer sb = new StringBuffer();  
sb.append("foot");    // "foot"  
sb.append("barn");    // "footbarn"  
sb.delete(3, 4);      // "foobarn"  
sb.replace(6, 7, "!!!"); // "foobar!!!"  
String result = sb.toString();
```

# Strings as Objects, cont.

- `int length()` - returns a `String`'s length as an `int` value:

```
z = new String("foobar!");  
int len = z.length(); // len now equals 7
```

- `boolean startsWith(String)` - returns `true` if the `String` to which this method is applied starts with the `String` provided as an argument, `false` otherwise:

```
// This will evaluate to true.  
String s = new String("foobar");  
if (s.startsWith("foo")) do something
```

# Strings as Objects, cont.

- boolean `endsWith(String)`

```
// This will evaluate to true.  
String s = new String("foobar");  
if (s.endsWith("bar")) ...
```

- int `indexOf(String)` - returns a non-negative int indicating the starting char. pos. (counting from 0) at which the String provided as an argument is found within the String to which this method is applied, or a negative value if the String argument is not found:

```
String s = new String("foobar");  
int i = s.indexOf("bar");    // i will equal 3  
int j = s.indexOf("cat");    // j will be < 0
```



# Strings as Objects, cont.

- `String replace(old char, new char)` - creates a new `String` object in which all instances of the character 'old' are replaced with the character 'new' - the original `String` remains unaffected:

```
String s = new String("o1o2o3o4");  
// Note use of single quotes around characters vs.  
// double quotes around Strings.  
String p = s.replace('o', 'x');  
// p now equals "x1x2x3x4"; s remains "o1o2o3o4"
```

# Strings as Objects, cont.

- boolean `equals(String)` - used to test the equality of values of one String object to another

```
String s = new String("foobar"); // two different obj.  
String p = new String("foobar"); // with same value  
if (s.equals(p)) ... // evaluates to true  
if (s == p) ... // IMPORTANT: evaluates to FALSE,  
                        // because == tests object identity
```

note, however:

```
String s = "foobar"; // pull from literal pool  
String p = "foobar"; // SAME OBJECT!  
if (s.equals(p)) ... // evaluates to true  
if (s == p) ... // now ALSO evaluates to true!
```

# Strings as Objects, cont.

- `String substring(int)` - creates a new `String` object by taking a substring of an existing `String` object starting at the position indicated by the `int` argument through the end of the existing `String`:

```
String s = "foobar";
```

```
String p = s.substring(3); // p equals "bar"
```

- `String substring(int, int)` - creates a new `String` by taking a substring of an existing `String` object starting at the position indicated by the first `int` argument and stopping just *before* the position indicated by the second `int` argument (we begin counting with 0 as the first character position):

```
String s = "foobar";
```

```
String p = s.substring(1, 4); // p equals "oob"
```

# StringTokenizer

- A handy class for splitting apart a String, based on a particular character(s) ("token")

```
// Must import java.util.*;  
String s = new String("This is a test.");  
StringTokenizer st = new StringTokenizer(s);  
while (st.hasMoreTokens()) {  
    System.out.println(st.nextToken());  
}
```

- Prints the following output:

```
This  
is  
a  
test.
```

# StringTokenizer, cont.

- May also specify a specific token character string when constructing the tokenizer:

```
// Must import java.util.*;
String s = new String("11/06/1956");
StringTokenizer st =
    new StringTokenizer(s, "/"); // note double quotes
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

- Prints the following output:

```
11
06
1956
```

# Exception Handling

- Exceptions are a way for the JVM to signal that a serious error condition has arisen during program execution

- Example: trying to access a non-existent object:

```
// We declare two Student object references,  
// but only instantiate one of these.
```

```
Student s1 = new Student();  
Student s2;
```

```
// This line of code is fine.  
s1.setName("Fred");
```

```
// This next line of code throws a  
// NullPointerException at run time.  
s2.setName("Mary");
```

# Exception Handling, cont.

- **Exception handling** enables a programmer to gracefully anticipate and handle such exceptions
  - Provides a way for a program to transfer control from within the block of code where the exception arose - known as the **try** block ...
  - ... into a special error handling code block known as the **catch** block


# Exception Handling, cont.

- The basics of exception handling are as follows:
  - We place code that is likely to throw an exception inside of a pair of curly braces, thereby turning it into a code block
  - We then place the keyword 'try' just ahead of the opening curly brace to signal the fact that we intend to catch exceptions thrown within that block
  - A 'try' block must be immediately followed by one or more 'catch' blocks
  - Each 'catch' block declares which category of exception it will catch, and then provides the 'recovery' code which is to be executed upon occurrence of that exception



# Exception Handling, cont.

```
Student s1 = new Student();
Student s2;
try {
    s1.setName("Fred");
    // This next line of code throws a
    // NullPointerException at run time ...
    s2.setName("Mary");
    // ... and as soon as the exception is detected by
    // the JVM, we automatically jump out of the
    // middle of the try block ...
    s1.setMajor("MATH");
    s2.setMajor("SCIENCE");
} // end of try block
catch (UnrelatedException e) { ... }
catch (NullPointerException e2) {
    // Recovery code for a null pointer exception.
    System.out.println("Darn!");
```



# Exception Handling, cont.

```
// continued from previous page

catch (Exception e3) { ... } // "catch-all" supertype
finally {
    // This code gets executed whether or not
    // an exception occurred.
    System.out.println("Finally ...");
}
```

# Exception Handling, cont.

- Note that the syntax of the catch block looks somewhat like a method, in that an argument is passed in (in parentheses)
  - This argument is passed in by the JVM, not by your own code -- i.e., we don't directly invoke a catch block the way we directly invoke a method

```
catch (NullPointerException e) { ... }
```

- The argument is always a reference to an object of some type of Exception -- we can "talk" to that object via dot notation to ask it to do certain things in helping us to diagnose the problem (see next page)

# Exception Handling, cont.

```
catch (NullPointerException e) {  
    System.out.println("Whoops!  An exception of type " +  
        e.getClass().getName() + " has occurred.");  
    System.out.println("Message:  " + e.getMessage());  
    System.out.println("Here is the stack trace:");  
    e.printStackTrace();  
    // and so forth  
}
```

# Exception Handling, cont.

- Try/catch blocks may be nested inside either the try or catch block of an outer try/catch

```
try {  
    open a user specified file  
}  
catch (FileNotFoundException e) {  
    try {  
        open a default file  
    }  
    catch (FileNotFoundException e2) {  
        attempt to recover ...  
    }  
}
```

# User-Defined Exceptions

- It is possible to define one's own exception types
  - These are useful in communicating error conditions among classes in our application
  - A user defined exception can be as simple as:

```
// MissingValueException.java
```

```
public class MissingValueException extends Exception {  
    // No need to override anything if you don't want to!  
}
```

# User-Defined Exceptions, cont.

- More common:

```
public class MissingValueException extends Exception {  
    // Constructor.  
    public MissingValueException(String message) {  
        super(message) ;  
    }  
}
```

# User-Defined Exceptions, cont.

- Also common:

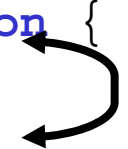
```
public class MissingValueException extends Exception {  
    // Define data structure for the custom exception.  
    Object someObject;  
  
    // Constructor.  
    public MissingValueException(String message) {  
        super(message);  
    }  
  
    // Extend.  
    public Object getObject() { ... }  
    public void setObject(Object someObject) { ... }  
}
```



# User-Defined Exception Classes

- Using our new exception type to signal an error condition back to the client code:

```
public class Student {  
    // Details omitted.  
  
    // We must declare that a given method throws an exception  
    // if we don't catch it within the scope of the method.  
    public void setName(String s) throws MissingValueException {  
        // We create a new instance when we throw it.  
        if (s.equals("")) throw new MissingValueException(  
            "A student's name cannot be blank");  
        else name = s;  
    }  
}
```



# Catching Thrown Exceptions

- In client code, the compiler will ensure that we either catch an exception ...

```
public class Course {  
    // Details omitted.  
    public void processStudent(Student st) {  
        read a value in from the GUI ...  
        try {  
            st.setName(s);  
        }  
        // The compiler will force you to catch  
        // this exception type because the setName()  
        // method declares that it is thrown.  
        catch (MissingValueException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

11/11/2009

# Catching Exceptions, cont.

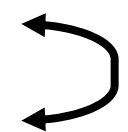
- ... or state that it is being thrown again
  - Effectively “passing the buck” to the client code that invokes this method

```
public class Course {  
    // Details omitted.  
    public void processStudent(Student st)  
        throws MissingValueException {  
        read a value in from the GUI ...  
        st.setName(s);  
    }  
}
```

# User-Defined Exception Classes

- We can throw many different types of exceptions from the same method:

```
public class Student {  
    // Details omitted.  
  
    public void setName(String s) throws MissingValueException,  
                                   InvalidCharacterException {  
        if (s.equals("")) throw new MissingValueException(  
            "A student's name cannot be blank");  
        else if (s contains an invalid character)  
            throw new InvalidCharacterException(s +  
                " contains a non-alphabetic character");  
        else name = s;  
    }  
}
```



# Catching Exceptions, cont.

- There is one subcategory of Exception called `RuntimeException` that the compiler doesn't "police" at compile time
  - Examples: `NullPointerException`, `ArithmeticException`
  - Typically, these represent design flaws that we should "bulletproof" our application against as we are building it

# Exceptions and Overriding

- When overriding a method that is declared to throw exceptions, we are permitted to alter the "throws" clause of the subclass as compared to the "throws" clause of the parent, but only in certain ways
- This is illustrated by an example on the following several pages

# Exceptions and Overriding

- Assume we have the following four exception types defined:

- The first three are unrelated:

```
public class XException extends Exception { }  
public class YException extends Exception { }  
public class ZException extends Exception { }
```

- The fourth is a subtype:

```
public class SubXException extends XException { }
```

# Exceptions and Overriding

```
public class Superclass {  
    // The parent class defines a method that throws  
    // two different types of Exception; neither type of  
    // Exception is a subtype of the other.  
    public void foo() throws XException, YException {  
        int x = 3;  
        if (x > 0) throw new XException();  
        else throw new YException();  
    }  
}
```



# Exceptions and Overriding

```
public class Subclass extends Superclass {  
    // We are going to attempt to override the foo() method  
    // in a variety of ways, to demonstrate what we can and  
    // cannot do in terms of the Exceptions that are thrown by  
    // the subclass.  
  
    // Attempt #1: omit an exception that the parent's  
    // version throws (namely, YException).  
    //  
    // THIS COMPILES AND EXECUTES PROPERLY.  
    public void foo() throws XException {  
        throw new XException();  
    }  
}
```

# Exceptions and Overriding

```
public class Subclass extends Superclass {  
    // We are going to attempt to override the foo() method  
    // in a variety of ways, to demonstrate what we can and  
    // cannot do in terms of the Exceptions that are thrown by  
    // the subclass.  
  
    // Attempt #2: add an exception that is unrelated to  
    // the types that the parent class throws.  
    //  
    // THIS DOES NOT COMPILE.  
    public void foo() throws XException, ZException {  
        details omitted ...  
    }  
}
```

# Exceptions and Overriding

```
public class Subclass extends Superclass {  
    // We are going to attempt to override the foo() method  
    // in a variety of ways, to demonstrate what we can and  
    // cannot do in terms of the Exceptions that are thrown by  
    // the subclass.  
  
    // Attempt #3: throw an exception that is a subtype of  
    // the types that the parent class throws.  
    //  
    // THIS COMPILES AND EXECUTES PROPERLY.  
    public void foo() throws SubXException {  
        throw new SubXException();  
    }  
}
```

# Exceptions and Overriding

- BOTTOM LINE: when you override a method that is declared by the parent class to throw exceptions, you are permitted to:
  - Eliminate some of the exceptions from the subclass version
  - Have the subclass throw a SUBTYPE of any of the exceptions that the parent throwsbut, you *MAY NOT* add an unrelated exception to the subclass's version of the method
- The reason for this is so that client code doesn't break as new subclasses are defined

# Exceptions and Overriding

- E.g., if I have the following client code (assume that myVector contains a random assortment of Superclass and Subclass objects):

```
for (int i = 0; i < myVector.size(); i++) {  
    // Always cast to a common supertype.  
    Superclass s = (Superclass) myVector.elementAt(i);  
    try {  
        s.foo();  
    }  
    // Since the foo() method, AS DEFINED IN THE  
    // Superclass class, is declared to throw exceptions  
    // X and Y, we must catch these ....  
    catch (XException x) { ... }  
    catch (YException y) { ... }  
}
```

# Exceptions and Overriding

```
for (int i = 0; i < myVector.size(); i++) {  
    // Always cast to a common supertype.  
    Superclass s = (Superclass) myVector.elementAt(i);  
    try {  
        s.foo();  
    }  
    // Since the foo() method, AS DEFINED IN THE  
    // Superclass class, is declared to throw exceptions  
    // X and Y, we must catch these ....  
    catch (XException x) { ... }  
    catch (YException y) { ... }  
    catch (ZException z) { ... }  
}  
  
// If a future subclass of Superclass were allowed to add  
// new exception types, the above client code would "break"  
// -- i.e., a ripple effect would have been created --  
// because we'd have to go back to the client code to add a  
// catch clause for the new type(s) of exceptions.
```

# Exercise #12

# "Wrapper" Classes for Simple Data Types

- We talked earlier about the fact that, prior to Java 5, Collection classes stored objects as "generic" Objects
  - Can we use a pre-5 Java collection such as an ArrayList to store simple data types, like int, double, or boolean?
  - Strictly speaking, no, because simple data types are not objects
  - There is a way around this problem, however, through the use of some of the other Java utility classes



# "Wrapper" Classes, cont.

- Java defines utility classes Boolean, Integer, Float, Double, etc. to correspond one-to-one with the simple data types
  - These classes can be used as object 'wrappers' around simple data types
  - Here is an example of how we'd use a Vector to store simple integers:

```
import java.util.*;

public class VectorTest {
    public static void main(String[] args) {
        Vector v = new Vector(); // of Integers
    }
}
```

# "Wrapper" Classes, cont.

```
// Let's store the integers 0 through 9 in a Vector.  
for (int i = 0; i < 10; i++) {  
    // We must build an object wrapper  
    // around each int value by calling the  
    // the Integer class's constructor,  
    // then add that wrapper object to the Vector.  
    v.add(new Integer(i));  
}
```

# "Wrapper" Classes, cont.

```
// Now, let's pull them back out.  
for (int i = 0; i < v.size(); i++) {  
    // Remember to (re)cast the object!  
    Integer objI = (Integer) v.elementAt(i);  
  
    // Pull the int back out of its wrapper.  
    System.out.println(objI.intValue());  
}
```

# Autoboxing in Java 5

- As of Java 5, collections *\*do\** support storing and retrieving simple data types

```
ArrayList<Integer> x = new ArrayList<Integer>();
```

```
// Adding simple int values ...
```

```
x.add(1);
```

```
x.add(2);
```

```
x.add(3);
```

```
// ... and retrieving them as simple int values.
```

```
for (int i : al) {  
    System.out.println(i);  
}
```

- Behind the scenes, they're automatically wrapped/unwrapped ("boxed/unboxed")

# "Wrapper" Classes, cont.

- The wrapper utility classes also have some static methods which are useful when you need to do data type conversions
  - For example, the Integer class defines a static method with signature:

```
int Integer.parseInt(String);
```
  - Pass in a String, and the Integer class will convert it to an int(eger) for you if it represents a valid integer, or will throw a NumberFormatException if it is not
  - This is useful when reading data from a GUI, because all data gets read in as Strings

# "Wrapper" Classes, cont.

- Here's an example of how this method can be used:

```
public class IntegerTest {  
    public static void main(String[] args) {  
        // Note shortcut way of creating an array.  
        String[] ints = { "123", "foobar", "456" };  
        int i = 0;  
        try {  
            for (i = 0; i < ints.length; i++) {  
                int test =  
                    Integer.parseInt(ints[i]);  
                System.out.println(test +  
                    " converted just fine!");  
            }  
        }  
        catch (NumberFormatException e) {  
            System.out.println(ints[i] +  
                " is an invalid integer.");  
        }  
    }  
}
```

11/11/2009

# "Wrapper" Classes, cont.

- This program, when run, produces this output:

```
123 converted just fine!  
foobar is an invalid integer.
```

- Here's another static method static defined on the Integer class:

```
String Integer.toString(int);
```

which will do the reverse: namely, turn an int into a String:

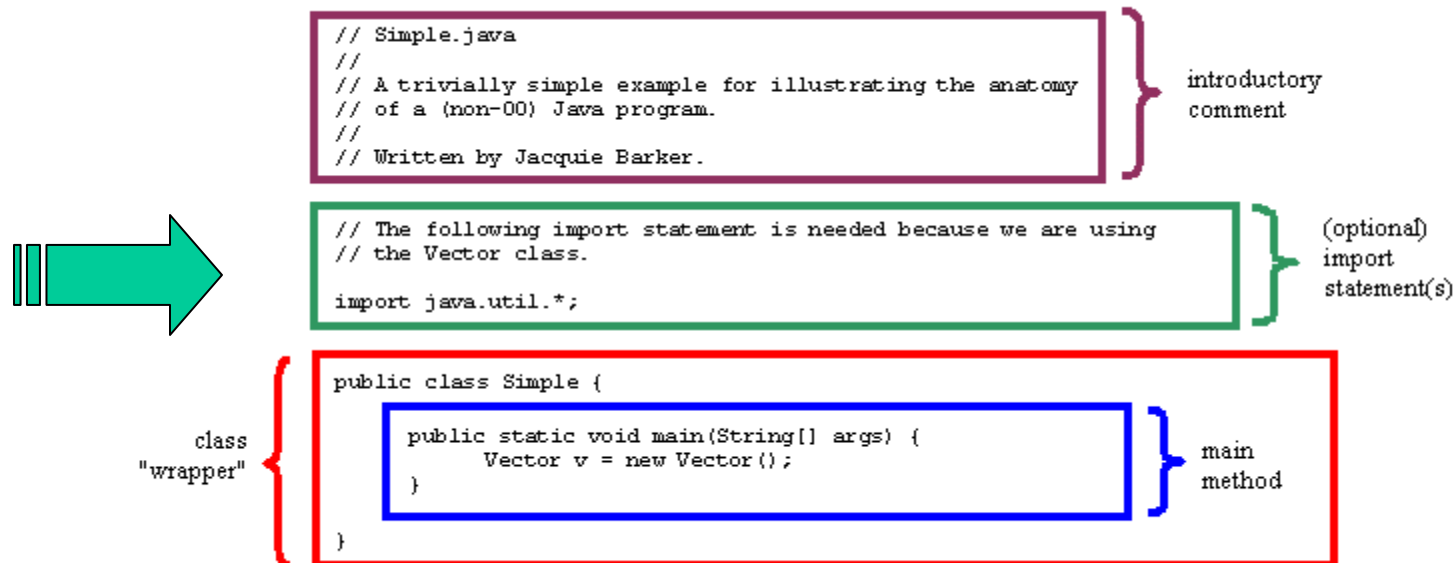
```
int j = 4;  
int i = 3 * j;  
String s = Integer.toString(i);
```

- There is a shortcut way for doing the same thing, however:

```
String s = "" + i;
```

# Import Statements

- When we originally discussed Java class/program anatomy, we skipped one key element: **import statements**, which we first discussed when talking about the Vector class





# Packages

- To fully appreciate import statements, we must understand the notion of Java **packages**
- Because the Java language is so extensive, its built-in classes are organized into logical groupings called packages
  - java.sql - classes related to communicating with ODBC-compliant relational databases
  - java.io - classes related to file I/O
  - java.util - utility classes, such as the Java collection classes that we discussed earlier
  - java.awt, javax.swing - classes related to GUI development

## Packages, cont.

- Most built-in Java package names start with 'java', but there are some that start with other prefixes, such as 'javax'
- If we acquire Java classes from a third party, they typically come in a package that starts with the company's domain name reversed, e.g. e.g. '**com.xyzcorp**.stuff' or '**gov.bls**.stuff'

# Packages, cont.

- The package named 'java.lang' contains the absolute core of the Java language
  - The classes contained within that package (e.g., String, Math) are always available to the compiler/JVM
  - We needn't worry about **importing** java.lang

# Classes in java.lang

Boolean

Byte

Character

Character.Subset

Character.UnicodeBlock

Class

ClassLoader

Compiler

Double

Float

InheritableThreadLocal

Integer

Long

Math

Number

Object

Package

Process

Runtime ←

RuntimePermission

SecurityManager

Short

StrictMath

String

StringBuffer

System ←

Thread ←

ThreadGroup

ThreadLocal

Throwable\*

Void

\* plus many derived Exception and Error classes

# Packages, cont.

- But, if we wish to instantiate an object of a type other than one of the classes in java.lang, then we import the package that it belongs to

```
// Simple.java
```

```
// Our class needs to instantiate a Vector,  
// and so we must import the package  
// that defines the Vector class.
```

```
import java.util.*;
```

```
public class Simple {  
    public static void main(String[] args) {  
        Vector v = new Vector();  
    }  
}
```

11/11/2009

Copyright © 2005 Jacquie Barker

709

# Packages, cont.

- The wildcard character (\*) at the end of an import statement informs the compiler that we wish to import *all* of the classes in a package
- As an alternative, we can import individual classes from a package:

```
// ImportExample.java

// Importing individual classes documents
// where each class that we are using originates.

import java.util.Enumeration;
import java.util.Vector;
import java.util.Date;
import java.io.PrintWriter;
// etc.
```

## Packages, cont.

- If we were to attempt to reference the Vector class in one of our classes *without* the proper import statement, we'd get the following compilation error:

`Class Vector not found.`

- This is because 'Vector' is not in the **name space** of our class: that is, it is not one of the names that the Java compiler recognizes in the context of that class

# Packages, cont.

- The name space for a given class contains:
  - the name of the class itself
  - the names of all of the features (attr./methods) of the class
  - the names of any local variables declared within any methods of the class
  - the names of all other classes belonging to the same package that the class in question belongs to
  - the names of all *public* classes in any other package that has been imported
  - the names of all public classes in java.lang
  - the names of all public features of any of the classes whose names are in the name space
  - Java reserved words



## Packages, cont.

- Importing a package is only effective for the particular .java file in which the import statement resides -- an import statement does not span an entire application
  - This is because each class is compiled separately, and hence has its own name space

# Packages, cont.

- An alternative to using import statements is to use **fully qualified** classnames in our code
  - I.e. we can prefix the name of a class with the name of the package from which it originates, as shown in the next example:

```
// Simple2.java
```

```
// No import statement!
```

```
public class Simple {  
    public static void main(String[] args) {  
        java.util.Vector v = new java.util.Vector();  
    }  
}
```

- Analogous to relative vs. full paths to files

# Programmer-Defined Packages

- Java provides programmers with the ability to logically group their *own* classes into packages
  - We could invent a package such as `com.objectstart.srs` to house our SRS application
  - In each of our classes that we wanted to include in the package, we'd add the line:

```
// Student.java
```

```
package com.objectstart.srs;
```

```
public class Student { ... }
```

# Programmer-Defined Pkg., cont.

- By convention, we organize the byte code (.class) files in a subdirectory hierarchy that parallels the package name
  - E.g., we want the Student.class file (and the byte code of any other classes belonging to the com.objectstart.srs package) to be stored in a folder hierarchy named `com/objectstart/srs`
  - We use the compiler command:  
`javac -d . *.java`  
to create this subdirectory automatically  
(demo - see directory exercise/soln/package, doit.bat and cleanup.bat)

# Programmer-Defined Pkg., cont.

- We typically bundle all classes in a given package into a compressed Java ARchive ("jar") file to distribute them
  - Like ZIP or Unix tar files
  - Continuing the previous example, if the home (sub)directory of the class files is "com" (because the package name is "com.objectstart.srs"), we'd use this command:

```
jar cvf SRS.jar com
```

(demo - see directory exercise/soln/package,  
doit.bat and cleanup.bat)

# Programmer-Defined Pkg., cont.

- Then, anyone else wishing to incorporate our SRS classes within an application that they were going to write would:
  - Include the import statement:

```
// MyApp.java

import com.objectstart.srs.*;
public class MyApp {
    Student s = new Student();
    // etc.
}
```

in their source code

# Programmer-Defined Pkg., cont.

- Reference our jar file in their classpath when compiling/running their application; e.g.,

```
javac -classpath SRS.jar;other_refs MyApp.java  
java -cp SRS.jar;other_refs MyApp
```

- Even though our compiled class files are kept *physically* separated from their application's compiled class files, our classes become *logically* combined with theirs within their application context

## Packages, cont.

- We didn't know anything about packages when we did the exercises for this course, and yet everything worked beautifully ... how is this possible?



# Packages, cont.

- If we do nothing to take advantage of programmer-defined packages, then:
  - As long as all of the compiled .class files for our code resides in the same directory on our computer system ...
  - They are automatically considered to be in the same package, known as the **default package**
  - All of the code that we wrote for the banking application was housed in the same directory (*exnn*), and hence fell within the same default package

# Packages, cont.

- This is what enables us to write code such as:

```
public class BankingApp {  
    public static void main(String[] args) {  
        BankAccount a = new BankAccount();  
        Person p = new Person();  
        // etc.  
    }  
}
```

without using import statements: `BankingApp`, `BankAccount`, and `Person` are all in the same (default) package

# Classpaths and jar Files

- Whenever we compile or run a Java program, the compiler or Java virtual machine needs to know where to look for the various classes that we are referencing in our code
- In this course, all of our lab exercises relied on code that was located in the default working directory, i.e., in the default package
  - The exception: all of the built-in Java classes, which are automatically found within the "Java Home" directory hierarchy (i.e., the hierarchy in which the Java compiler and JVM live)

# Classpaths and jar Files, cont.

- In an "industrial strength" application, however, this single-directory deployment strategy is too simplistic
- We often must access classes that are located in other directories, and/or that are stored in jar files
- To inform the compiler and JVM of where such classes are located, we can use one of two approaches

# Classpaths and jar Files, cont.

- Approach #1: we can establish an environment variable called `CLASSPATH` which lists all of the relevant directories/jar file locations

```
SET CLASSPATH=.;C:\foo\SomeJar.jar;D:\bar\a_directory
```

(where '.' is shorthand for the current working directory)

- Approach #2: we can set a command line flag for either the compiler or JVM, as we've previously seen

```
javac -classpath .;SomeJar.jar;AnotherJar.jar *.java
```

```
java -cp .;SomeJar.jar;AnotherJar.jar MyApp
```

# Classpaths and jar Files, cont.

- We can produce our own jar files via the DOS/Unix command line "jar" command:

`jar cvf name.jar files/directories to be included in jar`

e.g.

```
jar cvf MyJar.jar *.class *.java
```

- As we've seen, when a directory name is provided as input for a Jar file, the entire subtree beneath it is automatically included

```
jar cvf exercise.jar exercise
```

- Note that ANY file type can be stored in a jar: .class, .java, .gif, etc.

# Classpaths and jar Files, cont.

- To extract all files:

```
jar xvf name.jar
```

- To extract an individual file(s):

```
jar xvf name.jar filename(s)
```

e.g.

```
jar xvf srs.jar Student.java
```

- To see the contents of a jar: 

```
jar tvf name.jar
```

# Class Visibility

- We have two choices with respect to the manner in which we declare class visibility:

- As public:

```
public class Student {  
    // Attributes and methods ...  
}
```

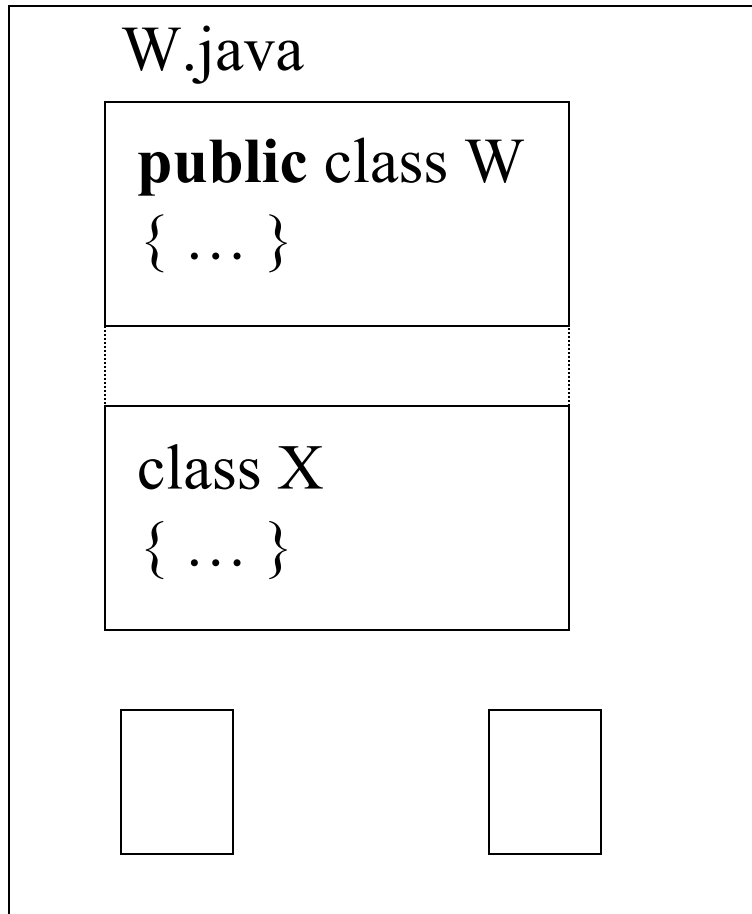
- With no explicit visibility:

```
class Professor {  
    // Attributes and methods ...  
}
```

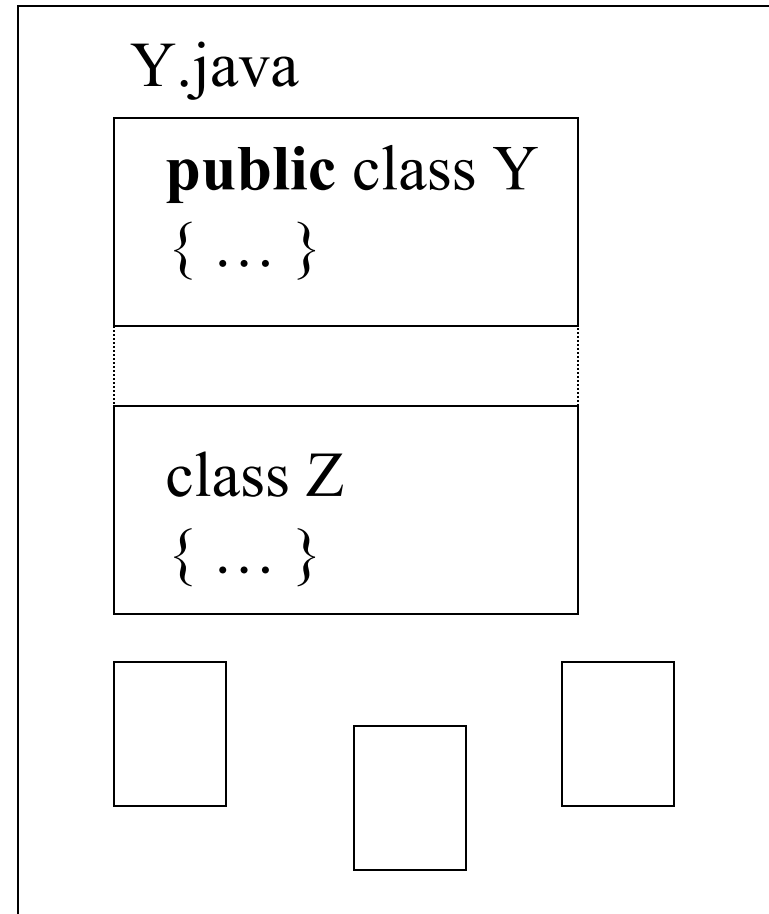
- The implications of these two choices are illustrated on the diagrams that follow



## Package A



## Package B



## Package A

W.java

```
public class W  
{ can create  
  objects of  
  type W and X,  
  but not of Y  
  or Z }
```

```
class X { ditto ... }
```

## Package B

Y.java

```
public class Y  
{ can create  
  objects of  
  type Y and Z,  
  but not of W  
  or X }
```

```
class Z { ditto ... }
```

## Package A

W.java

**import B.\*;**

public class W  
{ *can create  
objects of  
type W, X, **and Y**,  
but still not of Z* }

class X { *ditto ...* }

## Package B

Y.java

**public class Y**  
{ *can create  
objects of  
type Y and Z,  
but still not of  
W or X* }

class Z { *ditto ...* }

# Feature Visibility, Revisited

- We learned about three types of feature visibility early on: public, private, protected
- We hadn't talked about class visibility or packages at that time, and so we oversimplified what these three forms of visibility really entail - we'll revisit that now

# Feature Visibility, Cont.

- Also, if the access type for a given feature is omitted entirely, as in the following example:

```
public class Person {  
    public String name;  
    private String ssn;  
    protected String address;  
    int age;
```

then the feature in question is said to have **package visibility** by default

# Feature Visibility, Cont.


- Here is the effective visibility for a given feature of public class 'A' with respect to a class 'B', depending on where the two classes live:

(A = Person)

Specifier	Defining Class
private	✓
protected	✓
public	✓
default	✓

# Feature Visibility, Cont.

(A = Person) (B = Student)

Specifier	Defining Class	Subclass
private	✓	
protected	✓	✓
public	✓	✓
default	✓	

# Feature Visibility, Cont.

(A = Person) (Student) (B = Course)

Specifier	Defining Class	Subclass	Other Classes Same Package
private	✓		
protected	✓	✓	✓
public	✓	✓	✓
default	✓		✓

whoa!!!



# Feature Visibility, Cont.

(Person) (Student) (Course) (assuming import of x.y.Person)

Specifier	Defining Class	Subclass	Other Classes Same Package	Classes in Other Packages
private	✓			
protected	✓	✓	✓	
public	✓	✓	✓	✓
default	✓		✓	

# Feature Visibility, cont.

- For same-package classes, the bottom line is that if we don't explicitly declare something to be private, then it essentially becomes public:

```
public class Person {  
    // This first attribute is truly private.  
    private String ssn;  
  
    // But, all the rest of these attributes are  
    // effectively public from the perspective of  
    // other classes in the same package.  
    public String name;  
    protected String address;  
    // Exception: subclasses cannot see the next one!  
    int age;
```

# Determining the Class that an Object Belongs To

- Another method that all classes inherit from the Object class is:

```
Class getClass()
```

- The Class class, in turn, has a method with signature:

```
String getName()
```

- Used in combination, we can use these two methods to interrogate an object to find out what class it belongs to

# Determining the Class that an Object Belongs To, cont.

```
Student s = new Student();
Professor p = new Professor();
Vector v = new Vector();
v.add(s);
v.add(p);
for (int i = 0; i < v.size(); i++) {
    // Note that we are not casting the objects here!
    // We're pulling them out as generic objects.
    Object o = v.elementAt(i);
    if (o.getClass().getName().equals("Student")) {
        Student s = (Student) o;
        s.getGPA();
    }
}
```

11/11/2009

This program produces as output:

Copyright © 2005 by Ericnie Barber

740

Student

# Determining the Class that an Object Belongs To, cont.

- Another way to test whether a given object reference belongs to a particular class is via the `instanceof` operator

- This is a boolean operator which allows us to determine if some reference variable X is an object/instance of class Y

```
Student x = new Student();
```

```
if (x instanceof Professor) ... // will evaluate to false
```

```
if (x instanceof Person) ... // will evaluate to true
```

- The `classname` should be the fully qualified name of the class if it belongs to a package, e.g.,  
`java.lang.String`

# The toString() Method

- We mentioned earlier that all Java classes are descended from the Object class
- One of the features that all classes inherit from Object is a method with signature:

```
public String toString();
```

- As inherited, however, the method may not prove to be very useful, as the following example illustrates (cont.)

# toString(), cont.

```
Student s1 = new Student();  
s1.setName("Harvey");  
System.out.println(s1);  
// Prints out an object ID ... not too useful!
```

**Student@71f71130**

- It is a good idea to explicitly override the toString() method for any class that you invent

```
public String toString() {  
    return getName() + " [" + getSsn() + "];"  
}
```

```
// Prints "Fred" (or whatever).  
System.out.println(s1);
```

# Testing for Equality

- We've seen the use of a double equal sign (==) to test for equality of two values; e.g.,

```
int x = 3;  
int y = 4;  
if (x == y) do something ...
```

- We can also use == to test the "equality" of two objects:

```
Person p1 = new Person("Joe");  
Person p2 = new Person("Mary");  
if (p1 == p2) do something ...
```

- What does "equality" mean in the case of two object references?
- As we saw from our discussion of String as objects, the == operator tests to see if two references are referring to the same object



# Testing for Equality, cont.

```
public class Person {  
    private String name;  
  
    // Constructor.  
    public Person(String n) {  
        name = n;  
    }  
}
```

---

```
// Client code:  
Person p1 = new Person("Joe");  
Person p2 = p1; // Second handle on SAME object.  
Person p3 = new Person("Joe"); // New object, same NAME.  
  
if (p1 == p2) System.out.println("p1 equals p2");  
if (p1 == p3) System.out.println("p1 equals p3");
```

**Prints out:**                      p1 equals p2

# The equals() Method

- Another of the features that all classes inherit from Object is a method with signature:

```
public boolean equals();
```

- This method is used to test the "equality" of two objects in a different way: namely, based on whatever criteria that we establish by overriding the method
  - Let's look at an example

# The equals() method, cont.

```
public class Person {
    private String name;
    // Constructor.
    public Person(String n) {
        name = n;
    }
    // Overridden from Object.
    public boolean equals(Object o) {
        boolean answer = false;
        try {
            Person p = (Person) o; // Note cast.
            if (p.getName().equals(name)) answer = true;
            else answer = false;
        }
        catch (ClassCastException e) {
            answer = false;
        }

        return answer;
    }
}
```

11/11/2009

# The equals() method, cont.

```
public String getName() {  
    return name;  
}  
}
```

---

// Client code.

```
Person p1 = new Person("Joe");  
Person p2 = new Person("Mary");  
Person p3 = new Person("Joe"); // Different object, same name.  
Object o = new Object();  
  
System.out.println("Does p1 equal p2? " + p1.equals(p2));  
System.out.println("Does p1 equal p3? " + p1.equals(p3));  
System.out.println("Does p1 equal o? " + p1.equals(o));
```

Produces as output:

```
Does p1 equal p2? false  
Does p1 equal p3? true  
Does p1 equal o? false
```

# Reading from the Command Line

- Now that we've learned about Java Strings and arrays, we can appreciate how to pass data into a Java program when invoking it from the command line
- When we invoke a program, we can type data after the name of the program on the command line; e.g.

```
java Simple ABC 123
```

# Command Line Input, cont.

- Such data gets handed to the main() method as a String array called 'args' (or whatever else we wish to name it), as indicated by the main() method's argument signature:

```
public static void main(String[] args) { ... }
```

- Inside the main() method, we can do with 'args' whatever we'd do with any other array
  - E.g, determine its length, manipulate individual String items within the array, and so forth

# Command Line Input, cont.

```
public class FruitExample {  
    // If this program is run from the  
    // command line as follows:  
    //  
    // java FruitExample apple banana cherry  
    //  
    // then the args array will be automatically  
    // initialized with THREE String values  
    // "apple", "banana", and "cherry", which  
    // will be stored in array 'cells' args[0],  
    // args[1], and args[2], respectively.  
    public static void main(String[] args) {  
        // Let's print out a few things.  
        System.out.println("The args array contains "  
            + args.length + " entries.");  
    }  
}
```

# Command Line Input, cont.

```
// Only execute this next block of code if
// the array isn't empty.
if (args.length > 0) {
    int i = args.length - 1;
    System.out.println("The last entry is: "
        + args[i]);

    System.out.println("It is" +
        args[i].length() +
        " characters long.");
}

// etc.
}
}
```



## Command Line Input, cont.

- When this program is run from the command line as:

```
java FruitExample apple banana cherry
```

it produces the following output:

```
The args array contains 3 entries.  
The last array entry is: cherry  
It is 6 characters long.
```

# Accepting Keyboard Input

- Most applications receive information either directly from users via the application's graphical user interface, or by reading information from a file or database
- Until you have learned how to do such things with Java, it's handy to know how to prompt for textual inputs from the command line window

# Keyboard Input, cont.

- Just as Java provides a special OutputStream stream called System.out, which in turn provides both println() and print() methods for displaying messages to the command line window one line at a time ...
- ... Java also provides a special InputStream object called System.in to read inputs from the command line as typed by a user via the keyboard

# Keyboard Input, cont.

- There are two minor problems:
  - The primary method provided by the `InputStream` class for doing so, `read()`, only reads one character at a time, and
  - This character is actually returned to the code invoking `System.in.read()` as an `int(eger)` value
- Not to worry! We simply encapsulate (hide) these details in a class of our own making, called `Keyboard` (cont.)

# Keyboard Input, cont.

- The code for Keyboard.java, shown below, is provided in directory exercise\soln\keyboard

```
// Keyboard.java
// Copyright 2000 by Jacquie Barker - all rights reserved.
// THIS CLASS HAS BEEN TESTED UNDER WINDOWS/DOS, BUT NOT UNDER UNIX.

import java.io.*;

public class Keyboard {
    // Remember the last character typed, in both a char and
    // int form.
    private static char in;
    private static int in2;

    // The user's last full line of input gets saved in this String.
    private static StringBuffer keyboardInput;
```

# Keyboard Input, cont.

```
public static String readLine(boolean trimWhiteSpace) {  
    // Clear out previous input.  
    keyboardInput = new StringBuffer();  
  
    try {  
        // Read one integer, and cast it into a character.  
        in2 = System.in.read();  
        in = (char) in2;  
  
        // We are going to keep track of the previous  
        // character that has been typed, because  
        // on a DOS platform, pressing Enter generates two  
        // characters: a carriage return (13) followed by  
        // a linefeed (10). (Under Unix, it may only be a  
        // single linefeed character (10).)  
        int prevChar = -1;
```

# Keyboard Input, cont.

```
// Keep going until we encounter the value 10
// (line feed).
while (in2 != 10) {
    keyboardInput.append(in);
    prevChar = in2;
    in2 = System.in.read();
    in = (char) in2;
}

// If the last character appended was 13 (carriage
// return), remove it.
if (prevChar == 13) keyboardInput.deleteCharAt(
    keyboardInput.length() - 1);
}
catch (IOException e) {
    // details omitted ...
}
```

# Keyboard Input, cont.

```
String input = keyboardInput.toString();
```

```
// Strip off any leading/trailing white space, if desired.
```

```
if (trimWhiteSpace) input = input.trim();
```

```
// Return the complete String.
```

```
return input;
```

```
}
```

```
// An overloaded version of the readLine() method.
```

```
// By default, we trim white space with this version.
```

```
public static String readLine() {
```

```
    return readLine(true);
```

```
}
```

```
}
```



# Keyboard Input, cont.

- Here's an example of how to put the Keyboard class to use (this code is also found in `exercise\soln\keyboard`):

```
// KeyboardInputTest.java
// Copyright 2000 by Jacquie Barker - all rights reserved.

import java.util.*;
import java.io.*;

public class KeyboardInputTest {
    public static void main(String args[]) {
        // Prompt the user (note use of print() vs. println()).
        System.out.print("Enter your name: ");

        // Read one line's worth of input (up until user presses the
        // Enter key), trimming leading/trailing white space.
        String input = Keyboard.readLine();
    }
}
```

# Keyboard Input, cont.

```
// Display the input to verify the result.
System.out.println("You typed: |" + input + "|");

// Now, try the version that doesn't trim white space.

// Prompt the user (note use of print() vs. println()).
System.out.print("Enter white space, then press Enter: ");

// Read one line's worth of input (up until user presses the
// Enter key), preserving leading/trailing white space.
input = Keyboard.readLine(false);

// Display the input to verify the result.
System.out.println("You typed: |" + input + "|");
}
}
```

# Keyboard Input, cont.

- Running this program from the DOS/Unix command line would produce the following results (**bolded** text reflects that which was typed by the user, **b** represents a blank):

```
C:\> java KeyboardInputTest
Enter your name: bbJacquiebBarkerbbb
You typed: |JacquiebBarker|
Enter white space, then press Enter: bbxxxbb
You typed: |bbxxxbb|
```

# Reserved Words

<b>abstract</b>	default	goto	<b>null</b>	synchronized
<b>boolean</b>	do	<b>if</b>	<b>package</b>	<b>this</b>
<b>break</b>	<b>double</b>	<b>implements</b>	<b>private</b>	<b>throw</b>
<b>byte</b>	<b>else</b>	<b>import</b>	<b>protected</b>	<b>throws</b>
case	<b>extends</b>	<b>instanceof</b>	<b>public</b>	transient
<b>catch</b>	<b>false</b>	<b>int</b>	<b>return</b>	<b>true</b>
<b>char</b>	final	<b>interface</b>	<b>short</b>	<b>try</b>
<b>class</b>	<b>finally</b>	<b>long</b>	<b>static</b>	<b>void</b>
const	<b>float</b>	native	<b>super</b>	volatile
<b>continue</b>	<b>for</b>	<b>new</b>	switch	<b>while</b>