CS1110 Classes, stepwise refinement   18 Feb 2009

Miscellaneous points about classes.
More on stepwise refinement.

Next time:
Wrapper classes. Read
Section 5.1 of class text

Prelim 7:30-9:00 Thursday, 25 February

Review session: 1:00-3:00, Sunday, 21 Feb., Philips 101

1

Help: Get it now if you need it!!

• Call Cindy 255-8240 for an appointment with David Gries.

• See a consultant in the ACCEL Sun, Mon, Tues, Wed, Thurs 4:00pm to 11:00pm.

• See a TA.

• Peer tutoring (free). Ask in Olin 167 or visit
On http://www.engineering.cornell.edu, click on "student services". On the page that comes up, click on "Engineering Learning Initiatives (ELI.) " in the left column, upper part. Then, click on "peer tutoring" in the left column.

2

Content of this lecture

This lecture contains some final miscellaneous points to round out your knowledge of classes and subclasses. There are a few more things to learn after this, but we will handle them much later.

• Inheriting fields and methods and overriding methods.
  Sec. 4.1 and 4.1.1: pp. 142–145
• Purpose of **super** and **this**. Sec. 4.1.1, pp. 144–145.
• More than one constructor in a class; another use of **this**.
  Sec. 3.1.3, pp. 110–112.
• Constructors in a subclass —calling a constructor of the
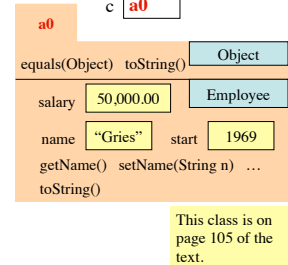 super-class. Sec. 4.1.3, pp. 147–148.

3

Employee c= **new** Employee("Gries", 1969, 50000);
c.toString()

Sec. 4.1,
page 142

Which method toString()
is called?

Overriding rule or
bottom-up rule:
To find out which is used,
start at the bottom of the
class and search upward
until a matching one is
found.

c **a0**

a0

equals(Object) toString()	Object
salary	50,000.00	Employee
name	"Gries"	start	1969
getName() setName(String n) … toString()

This class is on page 105 of the text.

Terminology. Employee **inherits** methods and fields from Object. Employee **overrides** function toString.

4

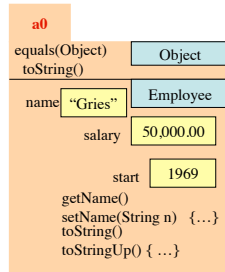**Purpose of super and this**
this refers to the name of the object in which it appears.
super is similar but refers only to components in the partitions above.

Sec. 4.1, pages
144-145

/** = String representation of this Employee */
public String toString() {
    return this.getName() + ", year " +
        getStart() + ", salary " + salary;
}

ok, but unnecessary

/** = toString value from superclass */
public String toStringUp() {
    return super.toString();
}

necessary

a0

equals(Object) toString()	Object
name	"Gries"	Employee
salary	50,000.00
start	1969
getName() setName(String n) {…} toString() toStringUp() { …}

5

A second constructor in Employee
Provide flexibility, ease of use, to user

Sec. 3.1.3,
page 110

/** Constructor: a person with name n, year hired d, salary s */
public Employee(String n, **int** d, **double** s) {
    name= n; start= d; salary= s;
}

First constructor

/** Constructor: a person with name n, year hired d, salary 50,000 */
public Employee(String n, **int** d) {
    name= n; start= d; salary= 50000;
}

Second constructor;
salary is always 50,000

/** Constructor: a person with name n, year hired d, salary 50,000 */
public Employee(String n, **int** d) {
    this(n, d, 50000);
}

Another version of second
constructor; calls first constructor

Here, **this** refers to the other constructor.
You HAVE to do it this way

6

1

public class Executive **extends** Employee {
  private double bonus;

/** Constructor: name n, year hired
       d, salary 50,000, bonus b */
public Executive(String n, **int** d, **double** b) {
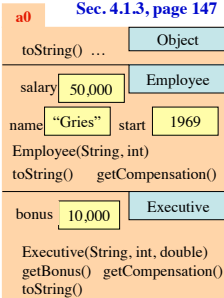   **super**(n, d);
    bonus= b;
  }
}

> The first (and only the first) statement in a constructor has to be a call to a constructor of the superclass. If you don't put one in, then this one is automatically used:
>
>   **super**();

Principle: Fill in superclass fields first.

Calling a superclass constructor from the subclass constructor
Sec. 4.1.3, page 147

a0

	Object
toString() …	

	Employee
salary	50,000
name "Gries" start 1969	

Employee(String, int)
toString()   getCompensation()

	Executive
bonus	10,000

Executive(String, int, double)
getBonus() getCompensation()
toString()

7

Anglicizing an Integer

anglicize("1") is "one"
anglicize("15") is "fifteen"
anglicize("123") is "one hundred twenty three"
anglicize("10570") is "ten thousand five hundred seventy"

/** = the anglicization of n.
    Precondition: $0 < n < 1{,}000{,}000$ */
public static String anglicize(**int** n) {

}

8

Principles and strategies

Develop algorithm step by step, using principles and strategies embodied in "stepwise refinement" or "top-down programming". READ Sec. 2.5 and Plive p. 2-5.

- **Take small steps**. Do a little at a time
- **Refine**. Replace an English statement (what to do) by a sequence of statements to do it (how to do it).
- **Refine**. Introduce a local variable —but only with a reason
- **Compile often**
- **Intersperse programming and testing**
- **Write a method specifications** —before writing the bodies
- **Separate your concerns**: focus on one issue at a time

9

Principles and strategies

• Mañana Principle.

During programming, you may see the need for a new method. A good way to proceed in many cases is to:

1. Write the specification of the method.

2. Write just enough of the body so that the program can be compiled and so that the method body does something reasonable, but no the complete task. So you *put off* completing this method until another time —mañana (tomorrow) —but you have a good spec for it.

3. Return to what you were doing and continue developing at that place, presumably writing a call on the method that was just "stubbed in", as we say.

10