CS1110 Stepwise refinement, more on classes 24 Sep 2009

Application of String processing and stepwise refinement.

Next time: Wrapper classes (Section 5.1)

Miscellaneous points about classes.

Prelim 7:30pm-9:00pm Thursday, 8 Oct (we'll announce which students should go to Olin 255 and which to Upson B17 later)

*If you have a conflict but did **not** receive an acknowledgment email yesterday*, email mwitlox@cs.cornell.edu ASAP to tell Maria Witlox what your conflict with the CS1110 prelim is.

Review session: 1-3pm Sunday, 3 Oct., Phillips 101

Have your iclickers out.

Reminders: finish revising A1s; A3 due Wed at midnight; pick up uncollected quizzes up front.

1

Help: Get it now if you need it!! Operators are standing by!!!

•See http://www.cs.cornell.edu/courses/cs1110/2009fa/staff.html for drop-in office hours and making appointments with Prof Gries, Prof Lee or any of the 5 graduate TAs;

or click the link at the bottom of that page to view the consultants' evening drop-in hours schedule.

• Free peer tutoring for engineering students is available: ask in Olin 167 or see http://www.engineering.cornell.edu/student-services/learning/peer-tutoring/index.cfm

2

iClicker continuous-feedback experiment

All through class, try updating us on your "status":

A. I get it, and you could speed up.
B. I get it, but don't go any faster, it's perfect.
C. I'm getting most but not all of it, but prefer that you just keep going anyway.
D. Could you repeat that just once more?
E. I'm totally lost.

3

An application: String processing, stepwise refinement

Strings are a particularly important data structure, because lots of information (especially non-numerical data) is stored in Strings.

For example, many webpages can, for many intents and purposes, be considered to be Strings.

Application: tracking live stock quotes from the Web.

4

Reminder: 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 method specifications** —before writing the bodies
- **Separate your concerns**: focus on one issue at a time

5

Odds and ends on classes: outline of remainder of lecture

More than one constructor in a class; another use of **this**. (Sec. 3.1.3, pp. 110–112)

Issues related to sub-classes:
- Inheriting fields and methods and overriding methods. Purpose of **super** and **this**. (Sec. 4.1 and 4.1.1: pp. 142–145)
- Constructors in a subclass —calling a constructor of the super-class. (Sec. 4.1.3, pp. 147–148)

[There are a few more things regarding classes to learn after this, but we will handle them much later.]

6

Why provide more than one constructor?

Sec. 3.1.3, page 110

Doing so is better for the user.
From A1 (we assume you remember the specs):

```
public Organism(int lev, int m, String nn) { …}
public Organism(int lev) { …}
```

So, the user can write new Organism(4) instead of new Organism(4, 0, null).

For the programmer,
it'd be great to have the one-parameter constructor call the other:

```
public Organism(int lev) {
   Organism(lev, 0, null);   this(lev, 0, null);
}
```

We wish we could say this!

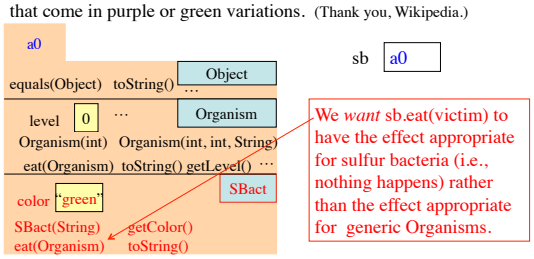
But you HAVE to do it "this" way.
Here, **this** refers to the other constructor.

7

Issues related to sub-classes

The ability to extend existing subclasses to reuse/refine existing behavior is a *terrific* aspect of object-oriented programming.

Example: modeling sulfur bacteria as photosynthesizing organisms that come in purple or green variations. (Thank you, Wikipedia.)

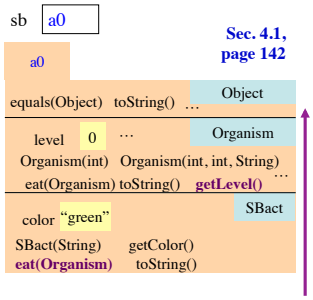


8

For the call sb.eat(v), which method eat is called?

Overriding rule or bottom-up rule:
Start at the bottom of the class and search upward until a matching method is found.

Sec. 4.1, page 142



Terminology. SBact **inherits** methods and fields from Organism. Sbact **overrides** eat and toString.

9

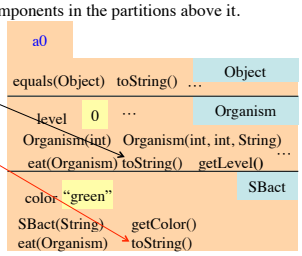
Purpose of super and this

Sec. 4.1, pages 144-145

Suppose we're overriding a method in order to modify it just a bit; so, we'd like to refer to the overridden method.
The word **super** refers only to components in the partitions above it.

```
/** = String like A3 requires for
Organism, but with "<green/
purple> SBact. " in front. */
public String toString() {
    return getColor()
      + " SBact. "
      + super.toString();
}
```

necessary

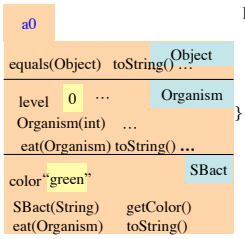


this refers to the name of the object in which it appears.
We could have written **this**.getColor(), but it wasn't necessary to.

10

Calling (reusing) a superclass constructor from the subclass constructor

Sec. 4.1.3, page 147




```
public class SBact extends Organism {
  private String color;
 /** Constructor: A sulfur bacterium of color c
[etc. Full program will be posted.]*/
  public SBact(String c) {
     super(0); /* default Org. values,
                * lowest level */
     color= c;
  }
}
```

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();

This corresponds to a natural principle: Fill in superclass fields first.

11