Application of String processing and stepwise refinement.

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

Next time: Wrapper classes (Section 5.1)

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

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.

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.


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

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.]
Why provide more than one constructor? From A1 (we assume you remember the specs):

```java
public Organism(int lev, int m, String mn) { ... }

public Organism(int lev) { ... }

Organism(lev, m, mn);  // This corresponds to a natural principle: Fill in superclass fields first.
```

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:

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

We wish we could say this!

## 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.)

```java
public Organism(int lev, String mn) {
  super(lev);  // We wish we could say this!
  color = mn;
}
```

For the call `sb.eat(victim)`, 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.

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

### Purpose of super and this

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.

```java
public String toString() {
  return getColor() + " SBact. The word super refers to the name of the object in which it appears.
  " + getLevel();
}
```

```java
public String eat(Organism victim) {
  return getColor() + " SBact. The word this refers to the name of the object in which it appears.
  " + getLevel();
}
```

We want `sb.eat(victim)` to have the effect appropriate for sulfur bacteria (i.e., nothing happens) rather than the effect appropriate for generic organisms.

```java
public String eat(Organism victim) {
  return getColor() + " SBact. The word super refers to the name of the object in which it appears.
  " + getLevel();
}
```

### Calling (reusing) a superclass constructor from the subclass constructor

Suppose we have a “green” color variant of the `Organism` class:

```java
public Organism(int lev) {
  super(lev);  // Here, this refers to the other constructor.
  color = "green";
}
```

This variant doesn’t need to override any methods; we can call its superclass constructor.

```java
public Organism() {
  super();  // This corresponds to a natural principle: Fill in superclass fields first.
}
```

The first (and only) 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:

```java
super();
```