Benjamin Fenster
CS-100 (Damon)
1 March 2003

Notes – Inheritance

- I. Nested Classes
  - a. Defined inside another class
  - b. Only the enclosing class can use it.
  - c. Must be private
  - d. Can call any method on enclosing class because it will be associated with an instance.
  - e. Code inside works just like it would outside except it also has access to the nested class's methods and representation.
- II. Anonymous Classes
  - a. A class doesn't always need a name; sometimes it's just a simple method or two.
  - b. An anonymous class must be a direct subtype of a named class or direct implementer of an interface.
  - c. It's defined as part of the 'new' construct
  - d. Works the same as a nested class except it cannot have a constructor.
  - e. Very, very common in GUI programs to stand-in for listeners.
  - f. Great for overriding just a method or two
    - i. MouseAdapter is an abstract class that provides empty methods for all the MouseListener. Only have to override the methods that aren't needed.
    - ii. Perfect for other Listeners since only a few methods are needed anyway.
  - g. Ugly, ugly syntax, but very useful.
- III. Rules for Inheritance
  - a. The Golden Rule
    - i. The most simple, working rule.
    - ii. All subtypes have all the behaviors of supertypes.
    - iii. That means subclasses can be used where a supertype was expected.
    - iv. That means methods don't have to know or care what type you've actually got.
    - v. Write code using the highest appropriate type you can (Object if possible.)
  - b. Signature Rule
    - i. Methods on the subclass need to have the same signature (return, throws, params) as on the superclass.
    - ii. Can accept more general arguments
    - iii. Can return more specific type.
    - iv. Cannot throw extra exceptions.
    - v. Signature must be exactly the same.
  - c. Methods Rule
    - i. Each method must do the same thing as on the superclass.
    - ii. If you know what the superclass does, the subclass's behavior shouldn't be surprising.
    - iii. Don't take over the method and do something completely different.
    - iv. "Not Surprising" is the way to think about it. Doesn't have to be *exactly* the same, but ought not to surprise anyone.
    - v. Methods do inherit pre- and post-conditions.
      1. Preconditions for the superclass must suffice for the subclass. Do not assume anything else is true.
      2. The same postconditions as on the superclass must be specified.
  - d. Property Rule
    - i. Any property that holds for the superclass must hold for the subclass.
    - ii. If Employee.salary() > Manager.salary() then EmployeeSub.salary() must be less as well.
- IV. Valid Uses of Inheritance
  - a. Specialize Behavior
    - i. Must, again, do something "not surprising."

ii. Ex: Point is general, CartesianPoint and PolarPoint are each more specialized.

b. Add Behavior

i. Sometimes truly new behaviors are added, more commonly just new state is added.

ii. State

1. Meeting adds attendees and location
2. Goal adds priority
3. Neither fundamentally changes the class, just adds new properties and their associated behaviors.

iii. A new behavior might be something like Meeting.checkConflict()

iv. Subclass Methods

1. May use superclass implementation

a. Easiest choice.

b. Just don't implement the method.

2. Can completely replace the implementation
3. Can extend the existing implementation.

a. Build that into whatever new code the method has.

b. Maybe just change the arguments and call the superclass method.

c. Call super.methodName()

d. In constructors, must call super() first since the object is not yet well-defined otherwise.

e. Ex: WeelyDisplay just manipulates its arguments and lets the parent constructor do the work.

f. Can always call static methods from constructors, but not instance methods.

V. Support in Superclass

a. Superclass needs to have the same behavior because the code should be written in terms of the superclass.

b. Very different levels of support can be provided though.

c. Concrete Class

i. Complete usable class by itself that happens to have a subclass.

ii. Maybe the subclass just needs to add new state, or maybe a complete new behavior.

iii. May include new representation, maybe not.

iv. Might hardcode behavior

1. That could render existing representation useless.
2. That can get dangerous.

v. Might reuse existing representation.

1. That could invalidate rep invariants.
2. REALLY dangerous, even if it's the right thing to do.

vi. Superclass can replace representation at any time, so subclasses should be wary of interpreting it differently.

vii. Ideal OO Language Idea:

1. Allow representation to be defined in "clumps"
2. Each package of representation could either be used as a whole or replaced as a whole.

d. Intermediate Class

i. Some behavior is fully implemented

ii. Some behavior is completely unimplemented.

iii. Some behavior is in between

1. Superclass may rely on methods that would need to be implemented by the subclass.
2. Any instance would be of a concrete subclass, so the methods would have to exist by the time they could be called.

    iv. Concrete subclasses would have to implement whatever behavior remains.
    v. Very few (if any) restrictions would be on the representation because none would exist already.

e. Interfaces

    i. Ultimate abstract class.
    ii. No representation, no behavior implemented.
    iii. Implementing an interface is easier than implementing from a superclass. No invariants to maintain, etc.
    iv. Example

        1. Iterator has an optional .remove() method that can throw an exception if implementation doesn't allow removing elements.
        2. Should have a subclass called RemovableIterator or something that would implement that more specialized behavior.