The UNIVERSITY of VERMONT

Benjamin Fenster
CS-100 (Damon)
15 February 2003

Notes – Inheritance

I. Concept
   a. Consider, as a metaphor, different boxes with buttons on them.
   b. A type describes a closed box – users outside can't see what's inside.
   c. Buttons on each box (which *are* visible to users) perform appropriate operations.
   d. Sometimes two boxes will have some of the same buttons, but not all.
      i. Create a new common type that includes just the buttons that are shared.
      ii. Every instance, even of subtypes, must satisfy the supertype's specification.
      iii. Supertype may or may not have instances.
   e. Hierarchy
      i. Supertype itself can have a supertype, and so on.
      ii. Higher up the hierarchy, classes are more abstract.
      iii. This in itself is a form of abstraction – very important!
      iv. Abstraction by specification: some details are passed down the tree; others are left out.
   f. The golden rule: *An instance of any type (whether of that type directly or of one of its subclasses) MUST satisfy all that type's specification!*
   g. Type hierarchy seems very natural
      i. Neanderthals were unable to generalize (a particular tool would be built out of only one material) but since Aristotle not much has changed.
      ii. Categorization and Generalization
      iii. Have to be able to forge the details when it's appropriate.

II. Type Definitions
   a. Definitions are never perfect.
   b. It's easy to find cases that violate the hierarchy (consider the platypus)
   c. In programming, the definitions *must not* have exceptions.
   d. Cannot have an instance that violates the specification of some higher-level type.
   e. Java Exceptions can be used to create more "breakable" specifications, but they then become part of the spec.
   f. Java Hierarchy
      i. `Object` is at the top.
      ii. Very little common representation
      iii. A way to determine an object's actual type is shared by all objects.
   g. Programming hierarchies are very broad and shallow – rarely more than few levels deep.
   h. Tend to have ragged bottoms – may go deeper in some places than in others.

III. Polymorphism
   a. "Many Shapes"
   b. Supertype has many different shapes of instances.
   c. Instances all look the same from one standpoint.
   d. Remember, all instances obey the supertype's specification.
   e. That means variables of a superclass type can hold an instance of a subclass!
   f. Variables and Assignments
      i. Declaring a variable of a supertype and using it to store a subtype is fine.
      ii. Declaring a variable of a subtype and using it to store a supertype is illegal.
   g. Subclasses do Four Things
      i. Add additional methods (new behaviors)
      ii. Replace an existing method with a more appropriate representation. (not as common)
      iii. Replace existing representation
         1. Can't, unfortunately, eliminate representation that's not needed anymore.
         2. CAN shadow it with new representation, but the storage for the old version is still allocated.

IV. Implementation

a. Eventually all this beautiful specification has to be reduced to simple code.
b. Subtypes that add representation need also add code for it.
c. Method Dispatching
    i. Find the "correct" code to execute at a particular moment.
    ii. Program needs to figure out which method to call (superclass? subclass?)
    iii. Only the most local representation can be called.
    iv. No access, by default, to the superclass methods.
    v. Even if the variable is a supertype, the actual type is determined at runtime (late-binding) and that method is used.
d. Multi-Method Dispatching
    i. Not test material.
    ii. Dispatching based on multiple arguments
    iii. Consider Printing as an example.
        1. Implementation depends on what type of document and what type of printer is being used.
        2. Different methods for pictures on a laser printer, text on a laser printer, text on a line printer.
    iv. Java doesn't allow multi-methods
    v. Must separate into two levels. (Document, Printer)
    vi. Document can call an appropriate method on Printer.
    vii. Problems with Multi-Methods
        1. Which class owns code that's a hybrid of two classes?
        2. M * N methods are needed instead of M + N
        3. Hard to understand for programmers.
e. Hierarchies
    i. Mostly language independent.
    ii. Subtype → Supertype is a fairly standard notation for denoting inheritance.
f. Coding in Java
    i. `extends` keyword. `public abstract class Fish extends animal`
    ii. (Abstract because no animal is JUST a fish. It must be a trout or a shark or something)
    iii. Some is exercised over subclasses
        1. Anything can be `final`
        2. Final classes cannot have subclasses
        3. Final methods cannot be overridden.
        4. Saying `final` in java is like NOT saying `virtual` in C++
        5. No further dispatching – final method gets all the calls from subclasses because you *cannot* re-implement it lower down!
g. Arrays
    i. Suppose Fish is a subclass of Animal
    ii. Then should `Fish[]` be a subtype of `Animal[]` ?
    iii. If so, a `Cat` could be inserted in a `Fish[]` and that's bad!
    iv. `Fish[]` should not be a subclass of `Animal[]`
    v. Java implemented it that way anyway, so now all array insertions have to be type-checked to make sure you `Cats` (in this example) are inserted.
h. Changing Specification in Subclasses
    i. It would be nice to change the specification later on, but not in Java/C++
    ii. You CAN tighten return types (make a method return something more specific than its parent)
    iii. You CAN loosen parameter types
        1. This is useful much more rarely than tightening return types.
        2. You can make functions more general though, which can be good.
    iv. Can, instead, write an additional method with the new functionality.

1. Write a copy of the original method, so that the superclass is overridden.
2. Write the new method the way you want, and CALL IT from the copy.

i. C++
   i. To replace implementation later, the function must be virtual
   ii. `virtual` is the anti-`final`
   iii. C++ makes the default NOT overriding.
   iv. Java defaults to overriding.
   v. In C++ once it's virtual, it's virtual forever. It's not possible to make it final midway down the hierarchy.

V. Abstract
   a. By default, classes are concrete. Concrete classes can be instantiated
   b. Classes can be made abstract, but to be useful they must have a concrete subclass.
   c. If you've got an abstract class, you can make an abstract method. All concrete subclasses MUST implement this method before it works.
      i. A concrete class might implement it directly.
      ii. An intermediate (abstract or not) class might do it.
   d. Abstract classes can have constructors and representation, which concrete subclasses can use.
   e. Can even call abstract methods, since the actual instance will definitely have access to real code.
   f. C++
      i. Can make an abstract class.
      ii. If any method is abstract, the class is.
      iii. There's no explicit declaration that makes a class abstract, so be sure to comment it!
      iv. Cannot make an abstract class without having an abstract method.
      v. All methods that are abstract must be virtual.
   g. Interfaces
      i. Interfaces are just abstract classes with all abstract methods.
      ii. Cannot have any representation, cannot have any implementation.
      iii. Classes use interfaces: `class MyClass implements MyInterface`
      iv. This allows a basic form of multiple inheritance,
         1. Interfaces can inherit from any number of interfaces.
         2. Classes can use any number of interfaces.
         3. Classes can inherit from no more than one class.
      v. Iterator is an interface
      vi. The name of an interface can be used anywhere the name of a class can be used.
      vii. `class A extends X implements Y, Z`

VI. Valid Uses of Inheritance
   a. Subclasses can do additional behavior
      i. Extending a concrete class, so existing functionality is already there.
      ii. Adding new behavior that's more specific than the parent class.
   b. Specialize behavior in a subclass
      i. Make it do something more specific.
      ii. Rectangle can do much more specific things than a Shape does.
   c. Implement existing behaviors.
      i. List has some behavior.
      ii. LinkedList and ArrayList implement those behaviors in a use-specific manner.
   d. Inherit Behavior
      i. (Especially common in C++)
      ii. Re-use old code
      iii. `class Roster extends List`

iv. It's possible to follow the specs, so that's not a problem.
v. It IS a problem that a Roster is not necessarily a List.
vi. Do you want Rosters to be accepted wherever Lists are accepted?
vii. It's much better to build a List *into* the Roster as private data.

e. Always think in terms of inheriting specification. If implementation comes with it, that's a bonus, but it should never be your goal.