Subclassing, method overriding, virtual methods

COMP 401, Fall 2018

Lecture 8

But first, some odds and ends

- Enumerations
  - Already seen them in A2
- Generics
  - Tomorrow's recitation

Motivating Enumerations

- Often need to model part of an object as one value from a set of finite choices
  - Examples:
    - Suite of a playing card
    - Day of week
    - Directions of a compass
- One approach is to use static named constants
  - lec7.ex6
- Drawbacks of this approach
  - No type safety
  - No value safety

Simple Java Enumerations

- General syntax:

  `access_type enum EnumName {symbol, symbol, ...};`

- Example:
  - public enum Genre {POP, RAP, JAZZ, INDIE, CLASSICAL}
- Enumeration name acts as the data type for the enumerated values.
  - Enumerated values available as *EnumName.symbol* as in: Genre.POP
- Outside of the class
  - Fully qualified name required as in: Song.Genre.POP
- Symbol names don't have to all caps, but that is traditional
- lec7.ex7

Enumerations in Interfaces

- Enumerations can be defined within an interface.
  - Useful when enumeration is related to the interface as an abstraction and will be needed by any/all specific implementations.
- lec7.ex8

Not so simple enumerations

- Java enumerations are actually much more powerful than this.
- Check out this tutorial for more: http://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html`

Generics

- A generic interface or class is one that is defined with respect to one or more “placeholder” reference types.
  - When used, you as the programmer must specify the specific type that replaces the placeholder.
  - Where by “reference type” I mean:
    - A class name
    - An interface name
- Generic interface/class names suffixed with a list of placeholder letters in angle brackets.
  - Examples: List<E>, Map<K,V>

List<E> and ArrayList<E>

- List<E>
  - An interface for anything that can act as a resizeable list of elements of type E
  - Main methods:
    - add(E element)
    - add(int index, E element)
    - E get(int index)
    - E remove(int index)
    - boolean remove(E element)
    - E[] toArray(E[] array_to_fill)
    - int size()
- ArrayList<E>
  - A specific implementation of List<E> that uses an internal array to store the elements.
- Lecture 8, Generics Example 1

Map<K,V> and HashMap<K,V>

- Map<K,V>
  - A "map" stores key-value pairs.
    - K is the type of key
    - V is the type of value.
  - Main methods:
    - V put(K key, V value)
    - V get(K key)
    - V remove(K key)
    - boolean containsKey(K key)
    - int size()
- HashMap<K,V>
  - Specific class that implements Map<K,V>
- Lecture 8, Generics example 2

Subclassing: A Motivating Example

- lec08.ex1.v1
- Suppose we're writing a university management system.
- Interfaces:
  - Person
    - get first and last name
    - get/set address
  - Student
    - add credits
    - get / set status (i.e., freshman, sophomore, junior, senior)
  - Professor
    - promote
    - get rank (i.e., assistant, associate, full)
- Classes:
  - StudentImpl implements Person, Student
  - ProfessorImpl implements Person, Professor

lec08.ex1.v1 Notes

- Student and Professor interfaces really should be subinterfaces of Person
  - Presumably implementing Student or Professor implies also being a Person
- lec08.ex1.v2

lec08.ex1.v2 Notes

- Casting no longer necessary
  - Anything that is a Student is also a Person
  - Anything that is a Professor is also a Person
- Notice how StudentImpl and ProfessorImpl implement the Person part of Student and Professor
  - Essentially the same implementation
    - Private fields for first, last, and address
    - Same definitions for Person methods
  - When two or more classes implement the same interface in the same way, then subclassing can help.

Extending Classes

- Declared with “extends” keyword
- Original class
  - Parent, parent class, superclass
- New class
  - Child, child class, subclass, extended class
- Subclasses inherit fields and methods from the parent class.
  - Purpose is to collect common implementation details from related classes into a single parent class.
  - Define each related class as a subclass that just adds the details that are not in common.

lec08.ex1.v3 Notes

- Notice parallel with interface structure
  - PersonImpl implements Person
  - StudentImpl extends PersonImpl
    - and implements Student which extends Person
  - ProfessorImpl extends PersonImpl
    - and implements Professor which extends Person
- Subclass constructor should call superclass constructor using “super”
  - Must be first line of subclass constructor
  - Alternatively, can chain to a different constructor that does.
  - If you don’t, then compiler will implicitly call super() with no arguments.

Subinterface vs. Subclass

- Extending interface only added behavior to contract.
  – Since interfaces don't specify (and don't care) how contract is fulfilled.
- Extending class creates a new class that shares internal implementation details of its super class.

```java
public class Bicycle {

   // the Bicycle class has three fields
   public int cadence;
   public int gear;
   public int speed;

   // the Bicycle class has one constructor
   public Bicycle(int startCadence,
               int startSpeed,
               int startGear) {
      gear = startGear;
      cadence = startCadence;
      speed = startSpeed;
   }

   // the Bicycle class has four methods
   public void setCadence(int newValue) {
      cadence = newValue;
   }

   public void setGear(int newValue) {
      gear = newValue;
   }

   public void applyBrake(int decrement) {
      speed -= decrement;
   }

   public void speedUp(int increment) {
      speed += increment;
   }

}
```

```java
public class MountainBike extends Bicycle {

   // the MountainBike subclass adds one field
   public int seatHeight;

   // the MountainBike subclass has one constructor
   public MountainBike(int startHeight,
                  int startCadence,
                  int startSpeed,
                  int startGear) {
      cadence = startCadence;
      speed = startSpeed;
      gear = startGear;
      seatHeight = startHeight;
   }

   // the MountainBike subclass adds one method
   public void setHeight(int newValue) {
      seatHeight = newValue;
   }
}
```

```java
MountainBike m = new MountainBike(10, 15, 0, 1);

m.setCadence(20);
m.setGear(2);
m.applyBreak(5);
m.speedUp(8);
m.setHeight(12);
```

http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

```
public class MountainBike extends Bicycle {
....
}
```

Extending a class is like writing a new class that has all the same details of the original class...

... plus adding additional stuff specific to subclass

```
public class MountainBike {
    public int cadence;
    public int gear;
    public int speed;

    public int seatHeight;

    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

Is-A Relationships for Subclasses

- Like interfaces, “is a” relationship is transitive up the subclass hierarchy.
  - A MountainBike object “is a” Bicycle
  - A StudentImpl “is a” PersonImpl
- Because you inherit everything your parent provides, by definition you also implement any interfaces your parent implements.
  - And so on all the way up the hierarchy.

Is-A For Subclasses

```
class A implements InterA {
...
}
```

Objects of type A, implement interface InterA.

A “is a” InterA

```
class B extends A implements InterB {
...
}
```

Objects of type B, implement interface InterB and InterA.

B “is a” A
B “is a” InterA
B “is a” InterB

```
class C extends B implements InterC {
...
}
```

Objects of type C, implement interface InterC, InterB, and InterA.

C “is a” A
C “is a” B
C “is a” InterA
C “is a” InterB
C “is a” InterC

Object

- All classes inherit from Object
  - Top of the class hierarchy.
  - Since every class must inherit from Object, don't actually need to specify it.

So when we say this:

```
public class MyClass {
...
}
```

We were implicitly doing this:

```
public class MyClass extends Object {
...
}
```

Object, cont'd.

- Because all classes implicitly have Object as a superclass ancestor...
  - A variable with data type Object can hold anything.
    - But then restricted to just the methods that are defined at the level of Object
- Public methods that all objects have:
  - public boolean equals(Object o)
  - public String toString()

Instance Fields

- Subclass has direct access to public and protected fields/methods of parents class, but not private ones.
  - Public: Everyone has access
    - Generally not a good idea.
    - Breaks encapsulation.
  - Private: Only class has access
    - Generally recommended as default.
    - Subclasses, however, also shut out.
  - Protected: Class and subclasses have access.
    - Like private (i.e., appropriate use of encapsulation) but allows subclasses to directly manipulate these fields.
- lec08.ex2

Access Modifier Chart

	Class	**Package**	**Subclass**	**World**
public	YES	YES	YES	YES
protected	YES	YES	YES	NO
no modifier	YES	YES	NO	NO
private	YES	NO	NO	NO

The dread pirate null…

- null is a legal value for any reference type variable.
  - Indicates a “lack” of value (i.e., points nowhere)
- Attempting to use a null reference, however, will result in program error
- Upshot: if a reference could possibly be null, you need to check it before using it.
  - Parameters passed to a method.
  - Result returned from a method.

Subclassing So Far

- A subclass inherits implementation details from its superclass
  - Fields
    - Direct access to public and protected fields
    - No direct access to private fields
  - Methods
    - Access to public and protected methods
    - No access to private methods
- Subclass constructors
  - Should call superclass constructor with super() as first line.
    - Or, chain to a different constructor
    - Or, rely on implicit call to super() constructor with no parameters.

Subclass Method Polymorphism

- Subclass can overload methods in superclass.
  - Remember, overloading is providing a different version of an existing method.
    - An example of polymorphism
    - Method signature is different in some way.
  - lec08.ex3

Overriding Methods

- A subclass can "override" a super class method by providing its own definition.
  - Method signature must be the same.
  - Original method must be visible from subclass
    - i.e., public, protected, or package-level access
- lec08.ex4

@Override directive

- So what's with the funny "@Override" line that Eclipse includes when generating a stub?
  - Known as a compiler "directive".
    - Completely optional, but useful
  - Indicates that the method is intended to override a superclass method.
    - Compiler will complain if it does not detect a visible superclass or interface method with the same method signature.
    - Helpful when you misspell a method name or attempt to override a method not visible to the subclass.
- lec08.ex5

Class Polymorphism

- Previously introduced the idea of “is-a” relationships
  - Between a class and interfaces implemented.
  - Between a class and its superclass hierarchy.
- This is also an example of polymorphism
  - Covariance
    - Treating an instance of a subclass (or interface) as an instance of the parent class (or interface).
    - This can be typed checked at compile type.
  - Contravariance
    - Treating a reference typed as the parent class (or interface) as an instance of a subclass (or interface).
    - Contravariance can not be type checked in advance at compile time.
    - Fails if the object is actually “invariant” with respect to the subclass.
- lec08.ex6, lec08.ex6main
  - Also demonstrates protected base class constructor

A Covariant Conundrum

- Problem:
  - What should happen when an overridden method is called on a covariant reference?

```
class A {
  public int m() {return 0;}
}

class B extends A {
  public int m() {return 1;}
}

class C extends B {
  public int m() {return 2;}
}
```

```
C c_obj = new C();
B b_obj = (B) c_obj;
A a_obj = (A) c_obj;

System.out.println(c_obj.m());
System.out.println(b_obj.m());
System.out.println(a_obj.m());
```

What should these lines print?

Solution 1: Non-virtual methods

- Let type of reference dictate which method definition is used.

```
class A {
  public int m() {return 0;}
}

class B extends A {
  public int m() {return 1;}
}

class C extends B {
  public int m() {return 2;}
}
```

```
C c_obj = new C();
B b_obj = (B) c_obj;
A a_obj = (A) c_obj;

System.out.println(c_obj.m());
System.out.println(b_obj.m());
System.out.println(a_obj.m());
```

If methods are non-virtual then these lines expected to print:

2

1

0

Solution 2: Virtual methods

- Use method defined by the actual type of object (even if reference is covariant)

```
class A {
  public int m() {return 0;}
}

class B extends A {
  public int m() {return 1;}
}

class C extends B {
  public int m() {return 2;}
}
```

```
C c_obj = new C();
B b_obj = (B) c_obj;
A a_obj = (A) c_obj;

System.out.println(c_obj.m());
System.out.println(b_obj.m());
System.out.println(a_obj.m());
```

With virtual methods, these lines expected to print:
2
2
2

Virtual Methods

- Different OOP languages choose to solve this problem in different ways.
  - C++, C#
    - Default is non-virtual solution.
    - Programmer can force virtual solution by marking a method with a special “virtual” keyword
  - Java
    - Methods are always virtual.
    - No special keyword needed.
- lec08.ex7

A virtual problem

- Drawback to the “always virtual” approach.
  - Consider the situation in which a subclass just needs a method to “do just a little more”.
    - In other words, wants to execute a method as defined in the superclass and then tweak the result.
    - Or maybe do something in advance of executing a method as defined in the superclass.
  - Because methods are always virtual, casting *this* reference to superclass in order access a parent class method won’t work.
- lec08.ex8 (won’t work)

It’s a bird, it’s a plane, it’s…

- … the *super* keyword.
- The super keyword provides exactly this ability to invoke methods on an instance as it is understood at the superclass.
  - Think of it as a version of “this” that is restricted to just what is provided by the superclass.
    - Note: Only goes up one level in class hierarchy
  - Essentially suspends “virtualness” of methods.
- lec08.ex9

Whence inheritance

- Related classes with common internals
  - Common fields used as part of methods with a common implementation.
  - Note, not just common behavior
- Specialization of existing classes after the fact.