Method Overload, Enumerations Inheritance, pt 1.

COMP 401, Spring 2016
Lecture 8
2/4/2016
Method Overloading

• Regular methods can also be overloaded
  – Same method name defined more than once.
• Return type may or may not be the same.
  – But usually is.
• Method type must be the same.
  – Instance method or static class method
  – Same access modifier
• Parameter list must somehow be different
  – Again, this is how the compiler knows which one is meant.
  – Either different in number or type (or both)
• One version can call another
  – No restrictions on when
  – No special syntax
• lec8.ex1, lec8.ex2
Why Overload?

- Provides access to constructor / method in a more context specific way.
- Limitations of overloading
  - Does not handle the case when you have two different situations that aren’t distinguished by the number or type of parameters being passed.
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 named constants
  – lec8.ex3

• Drawbacks of this approach
  – No type safety
  – No value safety
Simple Java Enumerations

- General syntax:
  
  ```java
  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

- lec8.ex4
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.

• lec8.ex5
Not so simple enumerations

• Java enumerations are actually much more powerful than this.

  – Also posted in quick links post on Piazza
Recap of Interfaces

• A “contract” for behavior.
  – Specifies a set of method signatures.
  – Defines a data type.
  – No implementation.
    • Specific classes implement the interface
Song and Video as Media

public interface Media {
    int getLengthInSeconds();
    double getLengthInMinutes();
    int getRating();
    void setRating(int new_rating);
    String getName();
}

public class Song implements Media {
    ....
}

public class Video implements Media {
    ....
}

We expect Song and Video to have methods matching those specified in Media.
Is-A and casting

• A class that implements an interface creates an “is-a” relationship between class data type and the interface data type.
  – A implements B => A “is a” B
    • Song is a Media
    • Video is a Media

• Casting allowed across an is-a relationship.

```java
Song s = new Song();
Media m;
m = (Media) s;

Video v = new Video();
Media m;
m = (Media) v;

Video v = new Video();
Song s;
s = (Song) v;
```
Inheritance

• What is inheritance in real life?
  – Characteristics / resources that you receive from your parents
    • Get these automatically.
    • Part of who you are.

• Similar idea in object-oriented programming.
  – In Java, concept of inheritance applied to both interfaces and classes.
    • Both signaled by the keyword “extends”
    • Similar in concept, but details are distinctly different.
      – Class inheritance more complex.
Extending Interfaces

• Adds methods to the contract.
  – Original:
    • parent interface, super interface
  – New:
    • subinterface, child interface, extended interface

• Created by using the “extends” keyword.

```java
public interface CompressedMedia extends Media {
    int getCompressedSize();
    int getUncompressedSize();
    Media uncompress();
}
```
Extension Creates Hierarchy

• Is-A relationship is transitive up the hierarchy.

public class Song implements CompressedMedia {
...

Song s = new Song();

CompressedMedia cm = (CompressedMedia) s; OK because s “is a” Compressed Media

Media m = (Media) s; OK because s is a Media by virtue of extension.

Song s2 = (Song) m; Casting from interface back to specific object type is allowed, but at runtime, if the object’s type does not actually match, a runtime exception will be thrown.
Extension vs. Multiple Interfaces

- Interface extension appropriate when additional methods make no sense without methods of the parent interface.
- Alternatively, can use multiple interfaces together as facets of an object.
Extension vs. Multiple Interfaces

public interface Compressed {
    int getCompressedSize();
    int getUncompressedSize();
    Media uncompress();
}

Instead of extending Media, Compressed is a separate interface and Song implements both.

public interface Media {
    int getLengthInSeconds();
    double getLengthInMinutes();
    int getRating();
    void setRating(int new_rating);
    String getName();
}

public class Song implements Compressed, Media {
    ...

    Song s = new Song();

    Media m = (Media) s;

    Compressed c = (Compressed) s;

    Song “is a” Media AND Song “is a” Compressed.
Subinterface Multiple Inheritance

• Multiple inheritance for interfaces is allowed.
  – A subinterface can extend more than one existing interface.
  – In this case, just a union of all methods declared in all of the parent interfaces.
    • Plus, of course, any additional ones added by the subinterface.

• lec8.ex6
Subclassing: A Motivating Example

- lec8.ex7.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
lec8.ex7.v1 Notes

• Student and Professor interfaces really should be subinterfaces of Person
  – Presumably implementing Student or Professor implies also being a Person

• lec8.ex7.v2
lec8.ex7.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.
lec8.ex7.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.
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;
    }
}

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;
    }

    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/landl/subclasses.html
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;
    }
}

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
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 {
    ...
}

class B extends A implements InterB {
    ...
}

class C extends B implements InterC {
    ...
}

Objects of type A, implement interface InterA.
    A “is a” InterA

Objects of type B, implement interface InterB and InterA.
    B “is a” A
    B “is a” InterA
    B “is a” InterB

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:                      We were implicitly doing this:

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

```java
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.

• lec8.ex8
## Access Modifier Chart

<table>
<thead>
<tr>
<th>Access Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>protected</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>no modifier</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>private</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>
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.
  - This might happen with methods that return null as a possibility.