Polymorphism, Inheritance Pt. 1

COMP 401, Fall 2014

Lecture 7

9/9/2014

Polymorphism

- Poly = many, morph = forms
- General principle of providing access to an abstraction or method in many forms
  - Idea is that different forms "fit" different contexts
  - Goal of the underlying functionality is the same.
- In OO programming, principle is evident in a number of different places.
  - Constructor overloading
  - Method overloading

Constructors

- What happens when you don't define a constructor.
  - Default constructor with no arguments.
    - Creates new object with all fields set to default value
      - Numeric fields set to 0
      - Boolean fields set to false
      - String, Array, and any other sort of reference value field set to null.
- lec7.ex1

An aside on putting more than one class in a file.

- A Java file can only have one “public” class.
  - A class marked public can be imported into other code.
  - Must match file name
- One or more “non-public” classes in a Java file can also be defined
  - Only available to code in the same package.
  - Does not have to match file name
  - Often used to define classes that are related to a public class in that file.

Constructor Overloading

- Can define multiple versions of the constructor.
  - Distinguished from each other by type and number of parameters
    - Must be some difference otherwise the compiler won't be able to tell them apart.
  - When you use the constructor, the right one will be chosen based on the parameters provided.
  - Note that once you define a constructor that takes parameters, the implicit, default no-argument constructor is no longer automatically available.
- lec7.ex2

Constructor Chaining

- Common pattern is to “chain” one constructor off of another.
  - To do this, the first line of code in the constructor must be the *this* keyword used as a function with parameters.
  - The matching constructor is called first and allowed to execute.
  - Then any subsequent code is executed.
  - Can chain multiple constructors one on to another
- lec7.ex3

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
  - 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
- lec7.ex4, lec7.ex5

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.

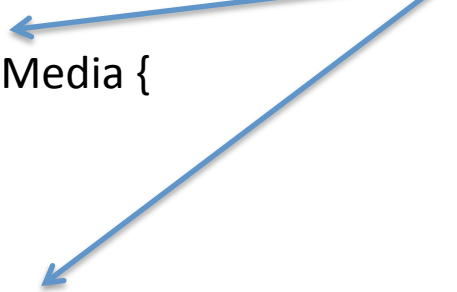
Recap of Interfaces

- A “contract” for behavior.
  - Defined by a set of method signatures.
  - Acts as 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();
}
```

We expect Song and Video to have methods matching those specified in Media.



```
public class Song implements Media {
....
```

```
public class Video implements 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.

```
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.

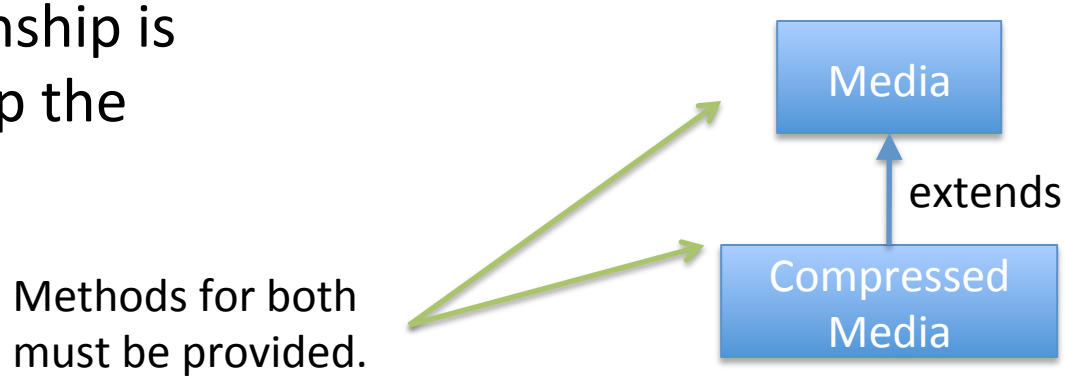
Extending Interfaces

- Adds methods to the contract.
  - Original:
    - parent interface, super interface
  - New:
    - subinterface, child interface, extended interface
- Created by using the “extends” keyword.

```
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. Composition

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

Extension vs. Composition

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

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

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

```
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.
- lec7.ex6

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
  - lec7.ex7
- 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.ex8

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 within any specific implementation.

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
  - Also posted on course resources page in Piazza.