

Polymorphism

COMP 401, Fall 2018

Lecture 07

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.
 - Today: Constructor overloading, method overloading, subinterfaces

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.
- lec7.ex1

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 if 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.ex2

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
- lec7.ex3, lec7.ex4

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.
 - Specifies a set of method signatures.
 - Defines a data type.
 - No implementation.
 - Specific classes implement the interface

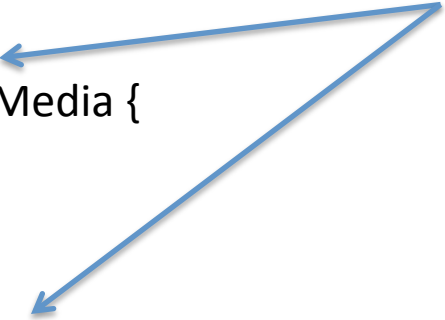
Song and Video as Media

```
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
- Assignment allowed across an is-a relationship.

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

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

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

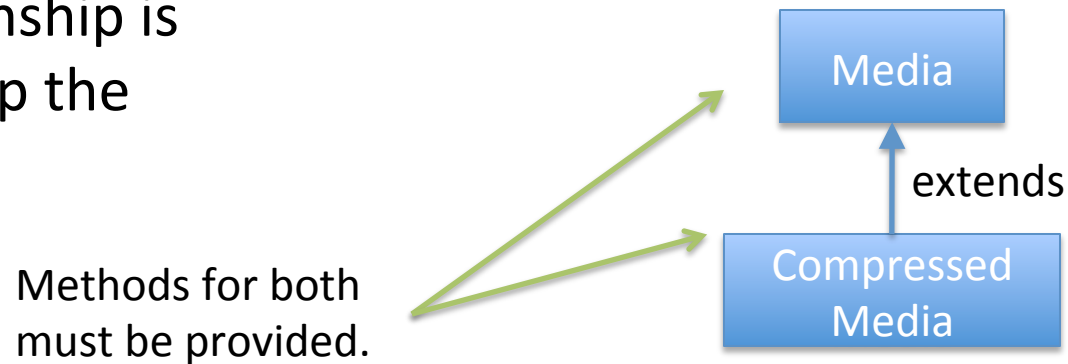
Extending Interfaces

- 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 = s;` *OK because s "is a" Compressed Media*

`Media m = 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 = s;
```

```
Compressed c = 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.
 - Useful when you want to program to a type that is a combination of interfaces.
 - Provides a type name for the combination.
- lec7.ex5