More on Interfaces

Chris La Pilla
CompSci 6
Summer 2007

Interfaces: Review

- Recall -- an *interface type* is a list of methods (without implementation)
- A class that *implements* the interface must include all of the methods in the interface
- A variable with an interface type can hold a reference to an object of *any* class that implements the interface

Example

- Recall `MeasurableShape` is an interface implemented by `Circle` and `Rectangle`
- Are the following lines of code legal?

  ```java
  Circle circle1 = new Circle(...);
  MeasurableShape shape = circle1;
  Circle circle2 = shape;
  shape = new Rectangle(...);
  shape = new MeasurableShape();
  ```

Converting between class and interface types

- Recall that we can assign a class type variable into an interface type variable:
  ```java
  MeasurableShape shape1 = new Circle(...);
  ```
- However, the opposite is not legal:
  ```java
  Circle circle2 = shape1; //syntax error
  ```
- We can convert from an interface type to a class type, but we need to use a cast
  ```java
  MeasurableShape shape1 = new Circle(...);
  Circle circle1 = (Circle) shape1;
  ```
- If `shape1` does not reference a `Circle` object, the program will throw an exception and terminate

Casting from interface types to class types

- Can convert from an interface type to a class type by casting:
  ```java
  MeasurableShape shape1 = new Circle(...);
  Circle circle1 = (Circle) shape1;
  ```

Casting

- The casting syntax is the same as for primitive types, but the meaning is different
  - Casting primitive types means that you are agreeing to a possible loss of information (e.g. truncating decimal)
  - Casting between interface types means that you are agreeing to take the risk of a possible runtime error
Overloading methods

- A class can contain more than one method with the same name, as long as the parameter list is different
- The correct version of the method is automatically chosen based on what the parameters are
- E.g. we can add a second constructor to Circle without the position parameters, and have it automatically set the Circle's position to (0,0)

Overloaded methods vs. polymorphic methods

- Recall polymorphism: when calling a method on an interface type, the actual method called is the one that belongs to the concrete class of the referenced object
- Both polymorphism and overloading involve deciding which method to call, but there is an important difference (see page 419)
  - Overloading: early binding (at compile time)
  - Polymorphism: late binding (at runtime – why?)